

Carlos A.S. Oliveira  
Panos M. Pardalos

# Mathematical Aspects of Network Routing Optimization

# Springer Optimization and Its Applications

---

VOLUME 53

---

## *Managing Editor*

Panos M. Pardalos (University of Florida)

## *Editor–Combinatorial Optimization*

Ding-Zhu Du (University of Texas at Dallas)

## *Advisory Board*

J. Birge (University of Chicago)

C.A. Floudas (Princeton University)

F. Giannessi (University of Pisa)

H.D. Sherali (Virginia Polytechnic and State University)

T. Terlaky (McMaster University)

Y. Ye (Stanford University)

## *Aims and Scope*

Optimization has been expanding in all directions at an astonishing rate during the last few decades. New algorithmic and theoretical techniques have been developed, the diffusion into other disciplines has proceeded at a rapid pace, and our knowledge of all aspects of the field has grown even more profound. At the same time, one of the most striking trends in optimization is the constantly increasing emphasis on the interdisciplinary nature of the field. Optimization has been a basic tool in all areas of applied mathematics, engineering, medicine, economics, and other sciences.

The series *Springer Optimization and Its Applications* series publishes undergraduate and graduate textbooks, monographs, and state-of-the-art expository work that focus on algorithms for solving optimization problems and also study applications involving such problems. Some of the topics covered include nonlinear optimization (convex and nonconvex), network flow problems, stochastic optimization, optimal control, discrete optimization, multi-objective programming, description of software packages, approximation techniques and heuristic approaches.

For further volumes:

<http://www.springer.com/series/7393>



Carlos A.S. Oliveira • Panos M. Pardalos

# Mathematical Aspects of Network Routing Optimization

Carlos A.S. Oliveira  
Research and Development Division  
Bloomberg LP  
New York, NY 10022  
USA  
[coliveira@gmail.com](mailto:coliveira@gmail.com)

Panos M. Pardalos  
Department of Industrial and Systems  
Engineering  
University of Florida  
303 Weil Hall  
Gainesville, FL 32611  
USA  
[pardalos@ufl.edu](mailto:pardalos@ufl.edu)

ISSN 1931-6828

ISBN 978-1-4614-0310-4

e-ISBN 978-1-4614-0311-1

DOI 10.1007/978-1-4614-0311-1

Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2011933566

Mathematics Subject Classification (2010): 47N10, 65K10, 49M25, 90C27

© Springer Science+Business Media, LLC 2011

All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed on acid-free paper

Springer is part of Springer Science+Business Media ([www.springer.com](http://www.springer.com))

# Contents

- 1 Unicast Routing Algorithms** ..... 1
  - 1.1 Unicast Routing Concepts ..... 1
  - 1.2 Flooding Algorithm ..... 2
    - 1.2.1 Implementation ..... 3
    - 1.2.2 Time-To-Live Window Strategy ..... 4
  - 1.3 Routing via Shortest-Paths ..... 5
  - 1.4 Distance-Vector Algorithms ..... 6
    - 1.4.1 The Bellman–Ford Algorithm ..... 6
    - 1.4.2 Implementation ..... 6
    - 1.4.3 Analysis of the Bellman–Ford Algorithm ..... 7
    - 1.4.4 Distributed Implementation ..... 8
  - 1.5 Link-State Algorithms ..... 8
    - 1.5.1 Dijkstra’s Algorithm ..... 9
    - 1.5.2 Distributed Implementation of Link-State Algorithms ... 9
  - 1.6 Additional Resources ..... 10
- 2 Multicast Routing** ..... 13
  - 2.1 Introduction ..... 14
    - 2.1.1 Definitions ..... 15
    - 2.1.2 Applications ..... 16
    - 2.1.3 Optimization Objectives ..... 16
  - 2.2 Multicast Technologies ..... 17
  - 2.3 Major Multicast Implementations ..... 19
  - 2.4 Basic Routing Techniques ..... 20
    - 2.4.1 Graph Theory Terminology ..... 20
    - 2.4.2 Flooding Algorithm ..... 21
    - 2.4.3 Algorithm Classification ..... 22
    - 2.4.4 Shortest Path Problems with Delay Constraints ..... 23
    - 2.4.5 Delay Constrained Minimum Spanning Trees ..... 23
    - 2.4.6 Center-Based Trees and Topological Center ..... 24

|          |                                                                  |           |
|----------|------------------------------------------------------------------|-----------|
| 2.5      | Additional Restrictions .....                                    | 25        |
| 2.5.1    | Nonsymmetric Links .....                                         | 25        |
| 2.5.2    | Delay Variation .....                                            | 26        |
| 2.6      | Additional Resources .....                                       | 26        |
| <b>3</b> | <b>Steiner Trees and Multicast .....</b>                         | <b>29</b> |
| 3.1      | Basic Concepts .....                                             | 29        |
| 3.2      | Steiner Trees on Graphs .....                                    | 30        |
| 3.3      | Modeling Multicast Routing Constraints .....                     | 31        |
| 3.3.1    | KMB Algorithm .....                                              | 32        |
| 3.3.2    | The KMB Algorithm on Multicast Networks .....                    | 33        |
| 3.3.3    | Other Algorithms for Steiner Tree .....                          | 33        |
| 3.4      | Steiner Tree Problems with Delay Constraints .....               | 34        |
| 3.4.1    | Adapting a Steiner Tree Algorithm .....                          | 35        |
| 3.4.2    | Algorithms for Sparse Steiner Trees .....                        | 35        |
| 3.4.3    | Combining Cost and Delay Optimization .....                      | 36        |
| 3.4.4    | Constrained Shortest Paths .....                                 | 37        |
| 3.4.5    | Link Capacity Constraints .....                                  | 38        |
| 3.4.6    | Delay Bounded Paths .....                                        | 38        |
| 3.5      | Sparsity and Delay Minimization .....                            | 39        |
| 3.5.1    | Sparse Groups .....                                              | 40        |
| 3.5.2    | Minimizing Configuration Time .....                              | 41        |
| 3.6      | An IP Formulation for Multicast Routing .....                    | 41        |
| 3.6.1    | Minimizing Bandwidth Utilization .....                           | 43        |
| 3.7      | The Degree-Constrained Steiner Problem .....                     | 44        |
| 3.8      | Conclusion .....                                                 | 45        |
| <b>4</b> | <b>Online Multicast Routing .....</b>                            | <b>47</b> |
| 4.1      | Introduction .....                                               | 47        |
| 4.2      | Online Multicast and Tree Recalculation .....                    | 48        |
| 4.3      | Speeding Up Routing Tree Computations .....                      | 48        |
| 4.3.1    | Working with High Complexity Algorithms .....                    | 49        |
| 4.3.2    | Random Graph Models .....                                        | 50        |
| 4.3.3    | Facilitated Inclusion of Group Members .....                     | 51        |
| 4.4      | Quality of Service Maintenance for Online Multicast Routing .... | 52        |
| 4.4.1    | Problem Definition .....                                         | 52        |
| 4.4.2    | Updating the Routing Tree .....                                  | 54        |
| 4.5      | OSPF-Based Algorithms for Online Multicast Routing .....         | 55        |
| 4.6      | Conclusion .....                                                 | 55        |
| <b>5</b> | <b>Distributed Algorithms for Multicast Routing .....</b>        | <b>57</b> |
| 5.1      | Distributed Algorithm Concepts .....                             | 57        |
| 5.2      | Distributed Version of the KMB Heuristic .....                   | 58        |
| 5.3      | Distributed Algorithm for Audio and Video on Multicast .....     | 59        |
| 5.4      | An Adaptation of Prim's Algorithm .....                          | 59        |
| 5.5      | Modifications of Dijkstra's Algorithm .....                      | 60        |

|          |                                                       |           |
|----------|-------------------------------------------------------|-----------|
| 5.6      | Using Flooding for Path Computation .....             | 61        |
| 5.7      | Algorithms for Sparse Groups .....                    | 62        |
| 5.8      | A Comparison of Distributed Approaches .....          | 62        |
| 5.9      | Conclusion .....                                      | 64        |
| <b>6</b> | <b>Center-Based Trees and Multicast Packing</b> ..... | <b>65</b> |
| 6.1      | Center-Based Trees .....                              | 65        |
| 6.1.1    | Center-Based Tree Concepts .....                      | 66        |
| 6.1.2    | The Topological Center Problem .....                  | 67        |
| 6.1.3    | Median and Centroid .....                             | 68        |
| 6.1.4    | Properties of Centroids .....                         | 69        |
| 6.1.5    | The CBT Protocol .....                                | 70        |
| 6.2      | The Multicast Packing Problem .....                   | 71        |
| 6.2.1    | Problem Description .....                             | 71        |
| 6.2.2    | Integer Programming Formulation .....                 | 72        |
| 6.3      | Multicast Dimensioning .....                          | 73        |
| 6.4      | Conclusion .....                                      | 75        |
| <b>7</b> | <b>Metaheuristics for Multicast Routing</b> .....     | <b>77</b> |
| 7.1      | Metaheuristics .....                                  | 77        |
| 7.1.1    | Genetic Algorithms .....                              | 78        |
| 7.1.2    | Tabu Search .....                                     | 80        |
| 7.1.3    | Simulated Annealing .....                             | 81        |
| 7.1.4    | Greedy Randomized Adaptive Search Procedure .....     | 81        |
| 7.2      | A GRASP for Multicast Routing .....                   | 82        |
| 7.2.1    | Introduction .....                                    | 82        |
| 7.3      | Basic Algorithm for the MRP .....                     | 84        |
| 7.3.1    | Adapting the Solution to the MRP .....                | 84        |
| 7.3.2    | Metaheuristic Description .....                       | 85        |
| 7.4      | Experimental Results .....                            | 92        |
| 7.5      | Conclusion .....                                      | 94        |
| <b>8</b> | <b>The Point-to-Point Connection Problem</b> .....    | <b>95</b> |
| 8.1      | Introduction .....                                    | 95        |
| 8.2      | Point-to-Point Connection and Multicast .....         | 95        |
| 8.3      | Problem Formulation .....                             | 96        |
| 8.3.1    | Mathematical Programming Formulation .....            | 98        |
| 8.4      | Asynchronous Team Algorithms .....                    | 99        |
| 8.4.1    | Formal Definition .....                               | 99        |
| 8.4.2    | Characterization of an A-Team .....                   | 101       |
| 8.5      | Solving the PPC Problem .....                         | 102       |
| 8.5.1    | Global Data Structures .....                          | 102       |
| 8.5.2    | Heuristic Strategies .....                            | 102       |
| 8.6      | A Partially Synchronous Distributed Model .....       | 105       |
| 8.6.1    | The Partially Synchronous Model .....                 | 105       |
| 8.6.2    | Distributed Memory Executions .....                   | 106       |

|           |                                                     |            |
|-----------|-----------------------------------------------------|------------|
| 8.6.3     | Types of Messages .....                             | 107        |
| 8.6.4     | Local Variables .....                               | 108        |
| 8.6.5     | Simulating the Partially Synchronous Model .....    | 108        |
| 8.7       | Computational Results .....                         | 111        |
| 8.7.1     | Hardware and Software Configuration .....           | 111        |
| 8.7.2     | Sequential Tests .....                              | 111        |
| 8.7.3     | Parallel Tests .....                                | 113        |
| 8.8       | Concluding Remarks .....                            | 115        |
| <b>9</b>  | <b>Streaming Cache Placement .....</b>              | <b>117</b> |
| 9.1       | Introduction .....                                  | 117        |
| 9.1.1     | Motivation .....                                    | 118        |
| 9.1.2     | Formal Description of the SCPP .....                | 118        |
| 9.1.3     | Related Work .....                                  | 119        |
| 9.2       | Types of Cache Placement Problems .....             | 120        |
| 9.2.1     | The Tree Cache Placement Problem .....              | 120        |
| 9.2.2     | The Flow Cache Placement Problem .....              | 122        |
| 9.3       | Complexity of the Cache Placement Problems .....    | 123        |
| 9.3.1     | Complexity of the TSCPP .....                       | 123        |
| 9.3.2     | Complexity of the FSCPP .....                       | 126        |
| 9.4       | Complexity of Approximation .....                   | 127        |
| 9.5       | Hardness of Approximation .....                     | 128        |
| 9.6       | Improved Hardness Result for FSCPP .....            | 130        |
| 9.7       | Conclusion .....                                    | 133        |
| <b>10</b> | <b>Algorithms for Cache Placement .....</b>         | <b>135</b> |
| 10.1      | Introduction .....                                  | 135        |
| 10.2      | An Approximation Algorithm for SCPP .....           | 136        |
| 10.2.1    | A Simple Algorithm for TSCPP .....                  | 136        |
| 10.2.2    | A Flow-Based Algorithm for FSCPP .....              | 137        |
| 10.3      | Construction Algorithms for the SCPP .....          | 139        |
| 10.3.1    | Connecting Destinations .....                       | 139        |
| 10.3.2    | Adding Caches to a Solution .....                   | 142        |
| 10.4      | Experimental Results .....                          | 145        |
| 10.5      | Conclusion .....                                    | 147        |
| <b>11</b> | <b>Distributed Routing on Ad Hoc Networks .....</b> | <b>149</b> |
| 11.1      | Introduction .....                                  | 149        |
| 11.1.1    | Graph-Based Model of Ad Hoc Networks .....          | 149        |
| 11.1.2    | Existing Algorithms .....                           | 150        |
| 11.2      | Ad Hoc Technology and Concepts .....                | 151        |
| 11.2.1    | Virtual Backbone Computation .....                  | 152        |
| 11.2.2    | Unit Disk Graphs .....                              | 153        |
| 11.3      | Maximum Connected Dominating Sets .....             | 153        |
| 11.3.1    | Existing Algorithms .....                           | 154        |

|           |                                                 |            |
|-----------|-------------------------------------------------|------------|
| 11.4      | An Algorithm for the MCDS Problem .....         | 154        |
| 11.4.1    | A Distributed Implementation .....              | 156        |
| 11.5      | Distributed Heuristic .....                     | 157        |
| 11.5.1    | Computational Complexity .....                  | 158        |
| 11.5.2    | Experimental Results .....                      | 160        |
| 11.6      | Conclusion .....                                | 162        |
| <b>12</b> | <b>Power-Aware Routing in MANETs .....</b>      | <b>163</b> |
| 12.1      | Introduction .....                              | 163        |
| 12.1.1    | Power Control Problem .....                     | 163        |
| 12.2      | MANETs and Power Management .....               | 164        |
| 12.2.1    | Resource Limitations .....                      | 164        |
| 12.3      | MANET Architecture .....                        | 165        |
| 12.4      | Related Work .....                              | 166        |
| 12.5      | Problem Formulation .....                       | 167        |
| 12.5.1    | An Integer Programming Model .....              | 168        |
| 12.6      | Variable Neighborhood Search Algorithm .....    | 170        |
| 12.6.1    | Neighborhood Descriptions .....                 | 171        |
| 12.6.2    | Distributed VNS Algorithm .....                 | 172        |
| 12.7      | Computational Experiments .....                 | 173        |
| 12.7.1    | Experimental Settings .....                     | 173        |
| 12.7.2    | Lower Bound on Power Consumption .....          | 174        |
| 12.7.3    | Integer Programming Results .....               | 174        |
| 12.7.4    | Heuristic Results .....                         | 174        |
| 12.8      | Conclusion .....                                | 175        |
| <b>A</b>  | <b>Concepts in Communication Networks .....</b> | <b>177</b> |
| A.1       | Computer Networks .....                         | 177        |
| A.1.1     | Network Topology .....                          | 178        |
| A.1.2     | Data Packets and Circuits .....                 | 178        |
| A.1.3     | Protocols .....                                 | 179        |
| A.1.4     | Protocol Stacks and Layers .....                | 180        |
| A.1.5     | Routing .....                                   | 181        |
| A.1.6     | Quality of Service .....                        | 182        |
| A.1.7     | Algorithmic Performance .....                   | 183        |
| A.2       | Main Algorithmic Problems .....                 | 184        |
| A.2.1     | Network Design .....                            | 184        |
| A.2.2     | Packet Routing .....                            | 185        |
| A.2.3     | Algorithmic Performance and Complexity .....    | 186        |
| A.2.4     | Heuristic Methods .....                         | 187        |
| A.2.5     | Distributed Algorithms .....                    | 189        |
| A.3       | Additional Resources .....                      | 189        |
|           | <b>References .....</b>                         | <b>191</b> |
|           | <b>Index .....</b>                              | <b>201</b> |
|           | <b>About the Authors .....</b>                  | <b>205</b> |



# List of Figures

|           |                                                                                                    |    |
|-----------|----------------------------------------------------------------------------------------------------|----|
| Fig. 1.1  | Pseudo-code for flooding algorithm.....                                                            | 3  |
| Fig. 1.2  | Pseudo-code for the Bellman–Ford algorithm.....                                                    | 7  |
| Fig. 1.3  | Retrieving a shortest path from the predecessor vector .....                                       | 7  |
| Fig. 1.4  | Pseudo-code for Dijkstra’s algorithm .....                                                         | 10 |
| Fig. 2.1  | Simple instance of multicast routing problem .....                                                 | 14 |
| Fig. 2.2  | Conceptual organization of a multicast group .....                                                 | 18 |
| Fig. 3.1  | Minimum spanning tree heuristic for Steiner tree .....                                             | 32 |
| Fig. 5.1  | Modification of Dijkstra’s algorithm for multicast routing, proposed by Shaikh and Shin [172]..... | 61 |
| Fig. 7.1  | Pseudo-code for genetic algorithm .....                                                            | 79 |
| Fig. 7.2  | Local search algorithm.....                                                                        | 80 |
| Fig. 7.3  | Algorithm for simulated annealing .....                                                            | 81 |
| Fig. 7.4  | Pseudo-code for GRASP .....                                                                        | 82 |
| Fig. 7.5  | KMB heuristic for the Steiner tree problem .....                                                   | 84 |
| Fig. 7.6  | Heuristic for the MRP .....                                                                        | 85 |
| Fig. 7.7  | GRASP construction phase.....                                                                      | 86 |
| Fig. 7.8  | Improved construction for GRASP .....                                                              | 87 |
| Fig. 7.9  | GRASP with path relinking for minimization .....                                                   | 89 |
| Fig. 7.10 | Procedure UpdateEliteSet .....                                                                     | 89 |
| Fig. 7.11 | Procedure Path Relinking.....                                                                      | 90 |
| Fig. 7.12 | Final Path Relinking .....                                                                         | 91 |
| Fig. 7.13 | Comparison between the average solution costs found by the KMB heuristic and our algorithm .....   | 93 |

|           |                                                                                                                                                                                                                              |     |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| Fig. 8.1  | Instance of the PPC. The <i>thick lines</i> represent the best solution, with value 44 .....                                                                                                                                 | 97  |
| Fig. 8.2  | Example of an A-Team. <i>Arrows</i> represent heuristics and the <i>rectangle</i> represents shared memory .....                                                                                                             | 100 |
| Fig. 8.3  | Organization of processes in the nodes of the parallel machine ....                                                                                                                                                          | 110 |
| Fig. 9.1  | Simple example for the cache placement problem .....                                                                                                                                                                         | 118 |
| Fig. 9.2  | Simple example for the tree cache placement problem .....                                                                                                                                                                    | 120 |
| Fig. 9.3  | Simple example for the flow cache placement problem.....                                                                                                                                                                     | 122 |
| Fig. 9.4  | Small graph $G$ created in the reduction given by Theorem 7. In this example, the SAT formula is $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee x_3 \vee \bar{x}_4) \wedge (\bar{x}_1 \vee x_3 \vee \bar{x}_4)$ ..... | 124 |
| Fig. 9.5  | Finding the optimal set of caches $R$ for a fixed tree .....                                                                                                                                                                 | 125 |
| Fig. 9.6  | Part of the transformation used by the FSCPP.....                                                                                                                                                                            | 127 |
| Fig. 9.7  | Example for transformation of Theorem 13 .....                                                                                                                                                                               | 131 |
| Fig. 10.1 | Spanning tree algorithm .....                                                                                                                                                                                                | 136 |
| Fig. 10.2 | Flow algorithm .....                                                                                                                                                                                                         | 138 |
| Fig. 10.3 | First construction algorithm for the SCPP .....                                                                                                                                                                              | 140 |
| Fig. 10.4 | Sample execution for Fig. 10.3 (Algorithm). In this graph, all capacities are equal to 1. Destination $d_2$ is being added to the partial solution, and node 1 must be added to $R$ .....                                    | 140 |
| Fig. 10.5 | Second construction algorithm for the SCPP .....                                                                                                                                                                             | 143 |
| Fig. 10.6 | Feasibility test for candidate solution .....                                                                                                                                                                                | 143 |
| Fig. 10.7 | Sample execution for Fig. 10.5 (Algorithm), on a graph with unitary capacities. Nodes 1 and 2 are infeasible, and therefore are candidates to be included in $R$ .....                                                       | 144 |
| Fig. 10.8 | Comparison of computational time for different versions of Figs. 10.3 (Algorithm) and 10.5 (Algorithm). Labels “C3”–“C10” refer to the columns from 3 to 10 on Table 10.2.....                                               | 148 |
| Fig. 11.1 | Approximating the virtual backbone with a connected dominating set in a unit-disk graph. The dashed line represents the transmission reach of transmitter $a$ .....                                                          | 150 |
| Fig. 11.2 | Heuristic for MCDS .....                                                                                                                                                                                                     | 154 |
| Fig. 11.3 | Compute a CDS .....                                                                                                                                                                                                          | 155 |
| Fig. 11.4 | Actions for a vertex $v$ in the distributed algorithm .....                                                                                                                                                                  | 159 |

Fig. 12.1 Graphical representation of an ad hoc network, where  
all wireless units have the same transmission reach.  
Nodes are linked by edges when the distance between  
them is at most 1 ..... 165

Fig. 12.2 Local search algorithm..... 170

Fig. 12.3 Variable neighborhood search algorithm..... 171

Fig. 12.4 Distributed algorithm for the PCADHOC ..... 173



# List of Tables

|            |                                                                                                                                                                                                                                                                      |     |
|------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| Table 5.1  | Comparison of algorithms for the problem of multicast routing with delay constraints .....                                                                                                                                                                           | 63  |
| Table 7.1  | Results of running the KMB heuristic and the GRASP metaheuristic on selected instances of the multicast routing problem.....                                                                                                                                         | 93  |
| Table 8.1  | Computational results comparing A-Team to a branch-and-cut algorithm for instances with 30 nodes .....                                                                                                                                                               | 112 |
| Table 8.2  | Computational results comparing A-Team to a branch-and-cut algorithm for instances with 40 nodes .....                                                                                                                                                               | 113 |
| Table 8.3  | Computational results comparing A-Team to a local search algorithm .....                                                                                                                                                                                             | 114 |
| Table 8.4  | Computational results comparing the speed up of a parallel A-Team with 2, 4, 8, and 16 nodes .....                                                                                                                                                                   | 114 |
| Table 10.1 | Computational results for different variations of Figs. 10.3 (Algorithm) and 10.5 (Algorithm). Times are given in seconds .....                                                                                                                                      | 146 |
| Table 10.2 | Comparison of computational time for Figs. 10.3 (Algorithm) and 10.5 (Algorithm). All values are in milliseconds.....                                                                                                                                                | 147 |
| Table 11.1 | Results of computational experiments for instances with 100 vertices, randomly distributed in square planar areas of size $100 \times 100$ and $120 \times 120$ , $140 \times 140$ , and $160 \times 160$ . The average solutions are taken over 30 iterations ..... | 160 |

|            |                                                                                                                                                                                                                                                                   |     |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| Table 11.2 | Results of computational experiments for instances with 150 vertices, randomly distributed in square planar areas of size $120 \times 120$ , $140 \times 140$ , $160 \times 160$ , and $180 \times 180$ . The average solutions are taken over 30 iterations..... | 161 |
| Table 12.1 | Results of computational experiments performed with the PCADHOC instances .....                                                                                                                                                                                   | 174 |

# Preface

We live in an increasingly connected world. Even before the appearance of the now ubiquitous broadband links and wireless systems, networks have been used to connect people in whole new ways. The technology currently available resulted in the creation of large scale, computational networked systems, such as the Internet. However, despite the impressive array of technologies developed for networking, efficient algorithms are more necessary than ever to solve some of the problems emerging in the area.

Algorithmic methods are necessary to provide logical connectivity among nodes of the network. Independent of topological and physical connections used, the goal of any communications system is to provide connectivity between its participants. In this context, general strategies such as *packet routing* have been devised to provide the necessary logical interconnection between nodes.

Routing algorithms create the illusion of a direct physical connection between networked nodes. The link is maintained despite the different media and equipment used to establish the real networking connection. It is this kind of logical connection that allows participants in a network to cooperate properly. Therefore, routing algorithms provide the basic means of communication among users of a networking system.

Since routing tasks must be continuously performed by networking equipment, it is also interesting to develop mathematical models to analyze their performance. It turns out that computing good solutions for routing problems is in many cases a difficult problem, for which only approximate or heuristic solutions are known in practice.

New mathematical models and techniques have been developed during the past decades to overcome such limitations. Many of the resulting models are based on linear and integer optimization methods, and assume that the network can be modeled as a system of nodes and links, with quantities associated only with these simple entities.

The result of the application of linear programming techniques is a powerful and flexible mathematical structure, capable of being manipulated by our algorithms. The resulting models can be used to provide insightful conclusions about the internal architecture and working of network routing.

## Multicast Networks

Multicast networks are an important type of communications system that has emerged in the last years to serve the increasing needs of network users. Multicast networks have several applications, such as real-time remote conferencing, web caching, software distribution, video-on-demand, work-group, and virtual-reality simulation.

Multicast networks can provide a very fast and reliable way to connect collaborating users, while avoiding unnecessary redundancy that can occur in traditional routing. All these features are made possible by the use of a different paradigm for data exchange: in such a network, data is viewed as a shared object that can be quickly and efficiently accessed by groups of interested users.

Multicast services are used in modern applications to allow direct communication between a source node and a set of destinations. In recent years, the number of applications of multicast has increased steadily, following the rapid advances of the Internet and local networks (intra-nets) in the corporate world.

A number of algorithmic issues, however, remain as a major problem for the wide deployment of multicast applications. For example, routing is an issue that has not been completely solved on multicast systems. While for traditional unicast systems the routing problem can be solved in polynomial time, using, for example, the Dijkstra's algorithm. The multicast routing problem is better modeled by the Steiner tree problem, which is known to be NP-hard [76, 185].

A multicast network has the main objective of allowing communication from a source to a set of destinations with one single send operation. This is made possible by retransmitting data whenever two or more destinations can be reached from one single node. A set of nodes interested in the same piece of data is called a *multicast group*. The main task faced in the operation of a multicast network consists of finding routes for delivering data to all members of a multicast group.

In contrast to the more traditional unicast systems (where there is only one source and one destination) [163], multicast networking involves the construction of routes that satisfy the requests of several customers at the same time. The resulting problems that need to be solved when routing on multicast systems are combinatorial in nature, and are usually hard to solve (from a computational point of view). To achieve good results, one frequently needs to apply sophisticated solution methods such as approximation algorithms, distributed computing, multi-objective optimization, and mathematical programming.

This book provides an overview of the mathematical and algorithmic results necessary for the solution of routing problems in unicast and multicast networks. It presents not only the mathematical tools, but also the main algorithms employed by researchers and practitioners working in the area.

## Objectives of the Book

In practice, several factors need to be considered when devising new computer networks, such as quality of the resulting links and their reliability. This book tries to give an overview of modeling techniques for such network routing problems. We are mostly concerned about computational issues arising in the process of optimizing network routes. Thus, the main objective will be deriving efficient algorithms, with or without guarantee of approximation.

The focus on algorithms is justified by considering that they are a cornerstone for the correct understanding of the protocols underlying multicast routing. The development of algorithms is also beneficial in several ways, such as improving the quality of provided service, reducing the cost of implementation and maintenance for existing networks, and increasing the overall bandwidth available for applications.

In the first part of the book, we provide a thorough introduction to the subject of algorithms for network routing, along with a discussion of basic techniques for algorithmic development.

In the second part of the book, we concentrate our discussion on two types of networks that have recently received much attention, due to their many applications. The first types are *multicast networks*, which are used to reliably share information on a group of clients. The second type of networks considered in this book are *wireless ad hoc systems*.

## Problems Considered

The field of network routing is very active and diverse, and we have no means of covering every aspect of it in this book. Due to the space limitations, we will concentrate our discussion on the following problems.

*Minimum cost routing:* A problem common to all protocols mentioned above is the necessity of finding routes that achieve some previously-specified criteria, such as cost, delay, or reliability. The achievement of these goals, however, requires the consideration of combinatorial sub-problems that are often difficult to solve efficiently. One example is the *minimum cost multicast routing problem* [156]. In this problem, we are given a network, a source node, a set of destination nodes, and costs associated with each network link. The objective of the problem is to find a set of links with minimum cost, while achieving the thresholds for other constraints

(such as delay and reliability). The problem can be easily seen as a generalization of the Steiner tree problem, which is NP-hard [76, 149].

Additional constraints of the minimum cost multicast routing problem involve time delay and reliability issues. For example, in most cases of the multicast routing problem it is required that data packets sent from the source reach destination nodes within a given delay threshold. Another important consideration on multicast networks is the reliability of the resulting system. For example, it is often advisable that each edge have utilization less than some threshold, to increase reliability [113, 119].

*Cache placement problems:* A second problem occurring in multicast networks is related to the determination of streaming cache locations. A multicast network requires *streaming cache* nodes whenever it is necessary to replicate the data received in one link to satisfy requests in different locations. A full scale multicast network is able to perform replication on every node, but this frequently results in additional cost (and processing overhead) in the whole network [126].

An alternative to a full scale multicast network is to locate cache nodes (splitters) in just a small subset of the network. This leads to savings for the organization maintaining the system, and offers a higher throughput at other nodes. The resulting *minimum cost cache placement problem* [144] asks for the location of a set of cache nodes with minimum cost. This problem has important economic consequences for institutions that rely on the deployment of multicast systems at low cost. However, the problem is very hard to solve, being related to the well-known set cover problem, which is also NP-hard [64].

## Audience

This book is targeted to graduate students, researchers, and professionals that may be interested in understanding the algorithmic and mathematical ideas behind routing in computer networks.

The presentation of concepts in this book is suitable for graduate students or even advanced undergraduate students. One of the general directions during the preparation of this book was in providing an overview of the methods and algorithms in network routing, specially when multicast is concerned. One of the ways of improving the understanding of the text was providing notes on some basic topics such as graph theory and linear programming. This will be useful for readers that are not fully acquainted to the mathematical topics necessary to understand the results presented throughout the book.

On the other hand, we tried to give enough information such that research topics could be explored, especially in the later chapters. Therefore, researchers in the area of network algorithms will also be interested in some of the most advanced topics discussed in this book. Moreover, we provide an extended bibliography that can be used as a starting point for access to more specialized sources.

## Book Organization and Topics

The topics in this book are organized as follows:

*Chapter 1:* In this chapter, we give an introduction to telecommunication networks, especially the concepts from unicast routing. These are problems that can be solved in polynomial time and therefore constitute a good introduction to the flavor of problems later explored in the text. Among the shortest-path methods, we present distance vector and link-state algorithms.

*Chapter 2:* This chapter presents the basic algorithms and models used in multicast routing. We cover multicast techniques and the main strategies used by them, such as the flooding protocol. At this point, we give a quick introduction to many of the topics that will be presented in later chapter.

*Chapter 3:* Steiner tree problems are very useful in representing solutions to multicast routing problems. They are employed mostly when there is just one active multicast group and the minimum cost tree is desired. In the Steiner tree problem, given a graph  $G(V, E)$ , and a set  $R \subseteq V$  of required nodes, we want to find a minimum cost tree connecting all nodes in  $R$ . The nodes in  $V \setminus R$  can be used if needed, and are called “Steiner” points. We discuss the main algorithms that use the Steiner problem as a model and how they are able to provide high quality solutions for such problems.

*Chapter 4:* Online versions of the basic routing problems are important for several reasons. The very fact that a computer network is an online system makes it necessary to devise methods for achieving good results in the presence of real time requirements. In this chapter we study some of the algorithms that have been proposed for multicast under the assumption of online operation.

*Chapter 5:* Similar to online requirements, distributed methods are commonplace in network applications. It is necessary to make sure that algorithms for multicast and unicast routing problems behave adequately when implemented in a distributed fashion. We present some of the most important distributed algorithms for multicast routing in this chapter.

*Chapter 6:* In this chapter, we explore the use of center-based trees for the determination of multicast routes. This method has been used in some applications, and it provides a good starting point when the number of multicast groups is large. There is a high potential for resource sharing in center-based algorithms, which can be explored to provide better solutions for traditional multicast routing problems. When considering multiple multicast groups, it is also necessary to make sure that the network possess enough capacity to satisfy all routes. Since this requires coordination among multiple network groups, we need to apply global techniques. We study the multicast packing problem in Chap. 6, especially the linear programming formulations used to make design decisions in this situation. We also consider the related multicast dimensioning problem.

*Chapter 7:* Many of the problems considered in this book are NP-hard. This indicates that it is hard to find exact solutions to most of these problems. On the other hand, the use of metaheuristics has demonstrated that good quality solutions may be found in reasonable time. In this chapter, we investigate how metaheuristics can be applied to problems in multicast routing. We present an implementation based on the greedy randomized adaptive search procedure (GRASP) metaheuristic, along with computational results of this approach.

*Chapter 8:* In this chapter, we discuss yet another problem related to multicast: the point-to-point connection problem. In this problem, we are given not only one source, but also multiple sources, and it is expected that the optimal routing tree (with minimum cost of connecting edges) is found. The problem is interesting in a situation where multiple caches of the same piece of information are maintained. We present some of the research done in this area using software agents to optimize the cost of the computed routes.

*Chapter 9:* We study in this chapter a problem in the area of multicast networks known as the streaming cache placement problem (SCPP). In the SCPP, one wants to determine the minimum number of multicast routers needed to deliver content to a specified number of destinations, subject to predetermined link capacities. We initially discuss the different versions of the SCPP found in multicast networks applications. Then, we show (using a transformation from the SATISFIABILITY problem) that many versions of the cache placement problem are NP-hard. Complexity results are then derived for the cases of directed and undirected graphs. We also consider different assumptions about the type of flow in the input network.

*Chapter 10:* In this chapter, we study some algorithms for solving the streaming cache placement problem. We present a flow-based algorithm that has an approximation guarantee in some special cases. We also describe heuristics for the cache placement problem, which can be customized for applications according to parameters supplied to the algorithm.

*Chapter 11:* In this chapter, we present an extension of the idea of multicast routing to wireless networks. The differences in transmission methods for wireless networks make it possible to have large gains in the computation of multicast trees. However, the dynamic nature of wireless networks also presents new challenges for the design of algorithms for efficient routing. We discuss some of the alternatives and show how existing algorithms perform in each situation.

*Chapter 12:* Finally, in this last chapter we discuss the connections between routing on wireless networks and power management for mobile devices. As power management is an important consideration for mobile systems, routing needs to adapt itself to the reduced power requirements. We present algorithms that can be used to generate routes while at the same time maintaining a low level of power consumption.

An Appendix is provided, with the most common terminology used on computer networks, distributed and wireless systems.

## Acknowledgments

This book was made possible only by the collaboration with several people working in the areas of optimization, algorithms, and telecommunications. Among them, Mauricio Resende (AT&T Research Labs) was instrumental in introducing some of these routing problems.

The first author thanks for the technical support, the School of Industrial Engineering and Management of the Oklahoma State University during the initial period of preparation of this book.

Thanks also to the students and professors of the Industrial Engineering department at University of Florida, where the research that resulted in this book started.

The manuscript and camera-ready copy for this book was created with free software. We used  $\text{\LaTeX}$  to do all the final typesetting. Most of the manuscript was prepared using `vim`, the free clone of `vi`. The `gcc` compiler was used to prepare C programs for some of the chapters. Finally, the working environment used includes many other UNIX utilities, all of them available at no cost.

New York, NY  
Gainesville, FL

Carlos Oliveira  
Panos Pardalos



# Chapter 1

## Unicast Routing Algorithms

Routing algorithms occur in diverse forms, which depend on the specific application for which they are employed. They constitute the foundation for networking protocols in the Internet and are deployed in millions of units of networking equipment, such as the in routers used throughout the world.

In this chapter, we will provide an introduction to some of the simplest routing algorithms, which are used for *unicast routing*. The main goal of this chapter is to provide a background that we can refer to when discussing the issues related to multicast and ad hoc networks.

In our discussion of algorithms, we will be concerned about time, space, and message complexity of these methods. The discussion of unicast routing algorithms will provide us with benchmarks for what is practically acceptable when working with more complicated problems, as we will see later in the book.

The simplest type of routing occurs when a node needs to send information to exactly one destination. This is what is called unicast routing. The interest in this type of routing strategy is twofold: first, it provides the basic ideas used in other routing techniques. Second, most of the well-known routing protocols are of this type, including the Internet Protocol (IP).

We introduce unicast routing using the simplest of all methods, called flooding. The flooding algorithm can be used to deliver information to any node in a network. We also present some of the shortest path based methods for network routing. These algorithms are generally classified as *distance-vector* or *link-state*. For each of these methods, we describe concrete implementations.

### 1.1 Unicast Routing Concepts

Among the existing algorithms to transfer data in a computer network, the most common are the ones employing *unicast routing* protocols. The main feature of these algorithms is that they are used to connect only two nodes: a source and a

destination, using a path that visits nodes in a predetermined set that corresponds to the location of routers.

From the optimization point of view, we can abstract the different nodes and links into a mathematical structure called a graph. A graph, denoted as  $G = (V, E)$ , is given along with a source  $s \in V$  and a destination  $d \in V$ . The objective is to find the best way of connecting  $s$  to  $d$ , using a minimum cost path.

Using a graph to represent the underlying structure, running a unicast routing protocol can be thought as similar to computing the shortest path algorithm on the underlying graph.

Traditionally, three main alternatives for the implementation of unicast routing have been devised. They are known as flooding, distance-vector routing, and link-state routing.

- *Flooding*: If the path from source to destination is not known, a routing method called flooding can be employed. The source node sends a packet to all destinations connected to it. Nodes receiving the message will either recognize the message as been destined to it, or will pass the message further along, until the required destination is reached.
- *Distance-Vector Routing*: With distance-vector routing, information about the distance from the current node to other nodes is stored in a table. A protocol for interchanging this routing information is needed to update the table after a given period of time to guarantee the accuracy of the distance information. The distance table is used to guide routers during the process of dispatching packets to their destinations. The link that offers connectivity at least cost is used to deliver packets.
- *Link-State Routing*: This is a type of algorithm for unicast routing where, given information about the connections in the network, each node computes the best routing to transfer information to any other destination node. This is done independently of other nodes, therefore, no data routing table needs to be shared among the nodes, as in the distance-vector routing protocol.

In the next sections, we provide a description of the main features of these routing methods.

## 1.2 Flooding Algorithm

Flooding is a method of transferring data between pairs of nodes that can be used whenever there is little or no information about the underlying topology of the network. The method consists primarily of passing data packets along, in such a way that all destinations for the packet are eventually reached.

The algorithm needs to maintain a local storage with information about packets that have been previously processed. This is to avoid situations where a packet is not

```

Flooding Algorithm
1  let  $v$  be the current node
2  let  $s$  be the source node
3  if  $v = s$  then
4      send the packet to all neighbors of  $v$ .
5  else
6      let  $R$  be the set of all packets processed by  $v$ 
7      let  $p$  be the packet received
8      let  $w$  be the neighbor of  $v$  that sent  $p$ 
9      if  $p$  is addressed to  $v$  then
10         accept  $p$ 
11     else
12         if  $p$  is in  $R$  then
13             do nothing, since  $p$  was already processed by  $v$ 
14         else
15             for each neighbor  $y$  of  $v$  such that  $y \neq w$  do
16                 send  $p$  to  $y$ 
17             end
18         end
19     end
20 end

```

**Fig. 1.1** Pseudo-code for flooding algorithm

reaching its destination, and therefore is returning periodically to the same nodes. This would eventually cause a congestion of the whole network, as routers would try to resend the same packet continuously.

### 1.2.1 Implementation

Although there are some variations, the basic algorithm can be described as shown in Fig. 1.1. One can see the basic computational decision in the flooding algorithm: checking if the received packet is addressed to the current node or to a different node in the network. In the first case, the data is simply accepted as it is and the process ends. However, if the destination is another node other than the current one, the algorithm will try to ensure that the information is sent to the correct destination. This basic process continues until all packets have been dispatched to their destinations.

The main advantage of flooding resides in being a very general algorithm, which is useful even in more difficult cases such as multicast routing, as will be seen later. Moreover, the basic technique used in the flooding algorithm can be employed without any previous information about the network topology. The fact that it uses only simple computational techniques also translates into easy of implementation.

Despite its usefulness, there are serious drawbacks in using the flooding algorithm as the only way of routing packets. The chief disadvantage of using this

method is that the resulting protocol may produce excessive traffic. Moreover, it is possible that it will use resources of the whole network to satisfy only a few nodes, when in practice a much better solution could be found.

There is also the problem of termination for the flooding protocol. Because the algorithm stores no information about the topology of the network, it cannot guarantee that packets will ever reach the specified destinations. Also, there is little information to determine if the packet has reached the destination or not. This translates into an indeterminate state for the transmission, since it is unknown if and when the data was received by the destination.

Finally, a big problem with flooding concerns the use of memory by client nodes. Notice that each router must keep a log of all packets it has seen to avoid infinite loops in transmission. As the number of transmitted packets increases, also increased is the necessary storage used by the each router. The flooding algorithm poses a heavy burden on the routing nodes, as they need to store data for each packet.

These undesirable features of the flooding routing algorithm makes it unsuitable for large scale routing from point to point. However, improvements have been proposed by several researchers. With some modifications, flooding has been used in the contexts of multicast routing and in mobile computing.

### ***1.2.2 Time-To-Live Window Strategy***

One way of avoiding the problems associated with flooding is to assign each packet a time to live (TTL) value, which is periodically decreased as the packet traverses the links in the network. With a TTL window, the routing algorithm can check general conditions such as if a packet has just recently left the origin, or if it has not reached the destination after several retransmissions.

The flooding protocol can be modified accordingly to decrease the TTL label for each package it transmits. Using this strategy, the algorithm will not resend any packet that has TTL equal to zero.

This solution, however, is not without its problems. First, it is not clear what could be the correct TTL for a given combination of packet and network. A bad combination of values can generate situations in which the decision taken by the protocol is not optimal, or even completely incorrect. For example, if the maximum TTL value is too small, it may expire before the destination is reached.

On the other hand, if the maximum TTL is too large, it can lead to wasted network resources. Routers may have to retransmit the same packet several times before it becomes clear that they cannot be sent to the required destination.

A possible solution is to require that the maximum TTL value be computed based on the topological characteristics of the network. Although this might be possible in some cases, it can further complicate the operation of the flooding protocol beyond what would be practical.

The discussion above shows that the flooding algorithm can be useful in some particular occasions, especially when no information is known about the topology of

the network. However, it cannot be used consistently to provide an efficient method for routing due to its shortcomings. In the next sections, we discuss more realistic algorithms that have been successful in practice.

### 1.3 Routing via Shortest-Paths

Routing protocols have traditionally relied in the computation of paths using algorithms referred to as *shortest path procedures* [70]. The main goal of these algorithms is, given a network of nodes and links, to compute the path with minimum cost connecting two predefined nodes. Cost in this context can be interpreted as any measure that is of interest for the application, such as distance or leasing costs.

Shortest path algorithms are some of the best known computational methods on networks. The pioneering work on algorithmic graph theory leading to the discovery of these methods was made during the fifties by [21] and [55]. The algorithms discovered at that time are still the basis for most of the protocols for unicast routing, which can be broadly divided into *distance vector* and *link state algorithms*. Both types of algorithms use at their foundation either the Bellman–Ford or Dijkstra shortest path algorithms.

A mathematical formulation of shortest path problems can be given in the following way. Let  $x$  be a vector in  $R^m$ , where  $m = |E|$ , and  $x_e$  is 1 if and only if the link  $e$  is selected as part of the shortest path. Then, the problem can be formulated as

$$\min c^T x$$

subject to

$$\sum_{j:(i,j) \in E} x_{ij} - \sum_{j:(j,i) \in E} x_{ji} = b_i \quad \text{for } i \in V$$

$$x \geq 0,$$

where  $b_i = 1$  if node  $i$  is either a source or a destination, and  $b_i = 0$  otherwise.

In this formulation, the objective function represents the total cost of the path. The vector variable  $x$  has entries that are either 0 or 1, such that the value of  $x_{ij}$  is 1 when edge  $(i, j)$  is selected for the shortest path.

The set of constraints require that the amount of flow leaving a node be equal to the amount of flow entering the node, plus or more the value of  $b_i$ . The value of  $b_i$  indicates if a particular node is a source or destination.

In this section, we present some of the main features of unicast protocols implemented using a shortest path algorithm. We will first present some implementation techniques based on the distance-vector algorithm. Then we will discuss link-state algorithms and how they can be applied to short path computation.

## 1.4 Distance-Vector Algorithms

An important class of shortest path solution methods is known as *distance-vector algorithms*. With distance-vector algorithms, a solution is created by computing a vector that records all distances from the source to the destinations. This vector is constantly updated until an exact solution for the shortest path problem is found.

### 1.4.1 The Bellman–Ford Algorithm

The best known example of a distance-vector method is referred to as the Bellman–Ford algorithm. This algorithm maintains a vector with information about the current known costs of the shortest paths between the nodes. The vector can be quickly initialized using the known costs of links (for pairs of nodes that are connected by an arc) or with a large value (in the case there is no arc between the nodes). To simplify the presentation, the shortest paths are initialized in a way such that all distances are infinity, while the distance from the source to itself is equal to zero.

During the operation of the Bellman–Ford algorithm, the distance-vector is updated using the following well-known property of shortest distances  $d(u, v)$  between nodes  $u$  and  $v$ :

$$d(u, v) \leq d(u, w) + c_{wv},$$

where  $c_{wv}$  is the cost of edge  $wv$ . That is, the shortest path distance between two nodes  $u$  and  $v$  is always less than the shortest path that passes through a third node  $w$ . Using this property, the algorithm simply tests each link in the network, updating the value  $d(u, v)$  whenever a node  $w$  is found that does not satisfy the above property. The value of the shortest path then becomes

$$d(u, v) = d(u, w) + c_{wv}.$$

### 1.4.2 Implementation

In Fig. 1.2, a pseudo-code for the Bellman–Ford algorithm is presented, showing the detailed steps. Two matrices  $D$  and  $P$ , of dimension  $n \times n$  (where  $n = |G|$ ), are used. Matrix  $D$  stores the known distances between the source  $s$  and each of the nodes in  $G$ . At the beginning, the only known distance is between  $s$  and itself, which is 0 by definition. We initialize the other distances with the value  $\infty$ .

Next, we perform the relaxation step  $n - 1$  times. For each such iteration, we look at each edge  $(u, v) \in G$  and update the quantity  $D[v]$  to guarantee that  $D[v] < D[u] + w(u, v)$ . If we need to update the distance, then we also update the predecessor  $P[v]$  to point to the node that is the previous in the minimum cost path.

```

Bellman-Ford Algorithm
1  let  $D$  be the distance vector
2  let  $P$  be a vector of predecessor nodes
3   $D[s] \leftarrow 0$ 
4   $P[s] \leftarrow \text{nil}$ 
5  for each vertex  $v \in G$  with  $v \neq s$  do
6     $D[v] \leftarrow \infty$ 
7     $P[v] \leftarrow \text{nil}$ 
8  end
9  for  $i \leftarrow 1$  to  $n - 1$  do
10   for each edge  $(u, v) \in E(G)$  do
11     if  $D[v] > D[u] + w(u, v)$  then
12        $D[v] \leftarrow D[u] + w(u, v)$ 
13        $P[v] \leftarrow u$ 
14     end
15   end
16 end

```

**Fig. 1.2** Pseudo-code for the Bellman-Ford algorithm

```

Algorithm Retrieve-Shortest-Path
1  let  $P$  be the predecessor vector
2  let  $w$  be the destination node
3  let  $\psi$  be a sequence of nodes
4  set  $\psi \leftarrow \emptyset$ 
5  set  $p \leftarrow w$ 
6  while  $p \neq \text{nil}$  do
7    add  $p$  to the beginning of  $\psi$ 
8     $p \leftarrow P[p]$ 
9  end

```

**Fig. 1.3** Retrieving a shortest path from the predecessor vector

At the end of the computation, vector  $D$  will store the costs of minimum paths from the source to each destination in the network. The predecessor vector  $P$  can later be used to retrieve the shortest path sequence itself, for each pair of nodes. One just needs to notice that, starting from any required destination  $w$ , the previous node leading to  $w$  is given by  $P[w]$ . Thus, the minimum cost path can be retrieved in reverse order, starting from the last node and traversing the path toward the origin. A pseudo-code for calculating this path is presented in Fig. 1.3.

### 1.4.3 Analysis of the Bellman-Ford Algorithm

The computational time complexity of the Bellman-Ford can be easily computed, by noticing that each edge is tested for each node of  $G$ . Thus, the number of iterations is  $O(nm)$ , where  $n = |V|$  and  $m = |E|$ . Notice that for dense graphs

the number of edges is  $\mathcal{O}(n^2)$ , making  $\mathcal{O}(n^3)$  the worst case of the algorithm. In practice, communication networks are sparse and the running time is more frequently given by  $\mathcal{O}(n^2)$ . Considering the simplicity of the algorithm, there are not many avenues for improvement of its running time, making this a tight analysis.

#### 1.4.4 Distributed Implementation

The main advantage of the distance-vector routing method is the relative simplicity of the underlying Bellman–Ford algorithm. Since there is no required order in which the edges must be tested by the main procedure, it becomes possible to perform the computation in a distributed fashion.

A distributed version of the Bellman–Ford algorithm is possible if each node maintains a shortest path vector – or at least the part of the vector that is required to compute the path in the local network. When the algorithm is implemented in a router, this distance-vector becomes part of the routing table. As adjacent edges are tested against the information stored in the routing table, the distributed algorithm sends regular updates about necessary changes in the known shortest paths.

The distributed version of the distance-vector routing algorithm implemented in this way will achieve optimality as soon as all nodes have sent the updated version of their distance information. Each node will have a copy of the results, based on the values found by other nodes during the distributed iterations.

A main disadvantage of the distance-vector method is that nodes are required to update information in the whole graph, increasing the number of required messages. Nonetheless, distance-vector algorithms have been used in real protocols such as routing information protocol (RIP) and Internet gateway routing protocol (IGRP) [136]. In general, distance-vector algorithms are used when the number of nodes is relatively small to avoid the overhead of routing table updates.

### 1.5 Link-State Algorithms

Another type of algorithm for constructing routes between the pairs of nodes is known as *link-state algorithm*. The main difference between link-state and distance-vector algorithms is that in the former each node needs to maintain a representation of the entire network to make routing decisions. Information about the connectivity of each node is exchanged, which allows nodes to construct a complete view of the network. After such information is collected, an algorithm is executed to determine the optimal next hop for each possible node.

Link-state routing protocols use underlying optimization algorithms to construct a routing table. One such algorithm is due to Dijkstra, which constructs the shortest

paths between two points. In the next sections, we describe the strategy used by Dijkstra's algorithm, along with the ways of improving its efficiency by the use of parallel processing.

### 1.5.1 *Dijkstra's Algorithm*

Dijkstra's algorithm is another method of finding the shortest path between any two nodes in a network. In fact, the Dijkstra's algorithm is capable of finding all shortest paths from a source to the rest of the network nodes. In this way, it is the ideal method to use when individual nodes need to compute a routing table for paths leading to other nodes.

Dijkstra's shortest path algorithm uses a vector to store the costs of shortest paths. Initially, all distances are marked as infinity (since they are unknown), with the exception of the source node itself, which has cost equal to zero. The algorithm also maintains a set of nodes for which the final shortest path cost is known. At the beginning of the computation, only the source is part of this set.

At each iteration, the algorithm looks at the neighborhood of the nodes that have been marked as completed. For each node in this neighborhood, the following is done: find the node with shortest distance (say  $v$ ), and mark it as a completed node. Then, looking at each neighbor  $w$  of  $v$ , check if the following constraint is satisfied:

$$d(w) \leq d(v) + c_{vw},$$

where  $c_{vw}$  is the distance between nodes  $v$  and  $w$ .

Whenever the constant is not satisfied, improve the shortest path to  $w$  using the new known value  $d(v) + c_{vw}$ . This is repeated until each node has been marked, after which the algorithm returns the vector of shortest paths.

For easy reference, the algorithm is shown in Fig. 1.4.

### 1.5.2 *Distributed Implementation of Link-State Algorithms*

To be useful as the basis for routing protocols, link-state algorithms need to be implemented in a distributed way. Each node, or at least a subset of the original network, needs to compute part of the link-state solution, and share this information with their neighbors.

This requirement is fulfilled by link-state algorithms, since the Dijkstra's algorithm has been readily implemented in a distributed way. The most common such implementation is embedded into current link-state protocols such as OSPF [134]. In the OSPF protocol, nodes share computed information about the connectivity state of the network, and use it to calculate the desired shortest paths.

**Fig. 1.4** Pseudo-code for Dijkstra's algorithm

```

Dijkstra's Algorithm
1   $A \leftarrow s$ 
2  for all  $v \in V$  do
3     $D[v] \leftarrow \infty$ 
4  end
5   $D[s] \leftarrow 0$ 
6   $found \leftarrow \text{false}$ 
7  while not  $found$  do
8     $v \leftarrow \arg \min_{u \notin A} D[u]$ 
9     $A \leftarrow A \cup \{v\}$ 
10   for all  $u \in N[v]$  do
11     if  $D[u] > D[v] + w(v, u)$  then
12        $D[u] \leftarrow D[v] + w(v, u)$ 
13     end
14   end
15   if  $v = t$  then
16      $found \leftarrow \text{true}$ 
17   end
18 end
19 return  $D[t]$ 

```

Another practical requirement of link-state protocols is that the computed information be shared regularly (at least every few hours). With this purpose, network nodes are instructed to periodically exchange known information.

Whenever connectivity data changes (this may be due to problems in a link or maybe the addition of new links), the connectivity map of the network will also change. These modifications are then passed on from the directly affected nodes to their neighbors. The collected data is shared using *link state advertisement*, that is, a periodical data update among routers, which provides information about recent changes occurred to links. Based on this connectivity data, shortest paths are then computed or modified.

## 1.6 Additional Resources

The area of single-source/single-destination routing is vast and touches many theoretical and practical disciplines or computing. We list a few reference works in the area, to give the reader a quick view of the existing literature.

Starting with the theory of networks, one can refer to Ahuja et al. [3], which is a standard reference on network optimization methods. This book contains not only algorithms for telecommunications problems, but also for network applications in general. A similar text, but with a more concise treatment of network optimization theory is available by Cook et al. [40].

Papadimitriou and Steiglitz [147] is a classical text exposing the theory of polynomial algorithms for combinatorial optimization. It has a wealth of theory and exercises in network optimization. Other books of notice in this area include [20] and [139].

It is also important to mention some of the literature on practical implementations of unicast routing. A very clear treatment of OSPF is available in Moy [136] (the author of that book has also prepared one, the RFCs defining OSPF [134]). The most up to date information in protocol matters can be accessed on the web site of the Internet Engineering Task Force (IETF, at <http://www.ietf.org>), an organization that publishes standards and documentation for most of the TCP/IP-related protocols.

Details about link-state and distance-vector algorithms discussed in this chapter are available in the form of RFC documents (Request for Comments) in the IETF web site. These documents describe protocol implementation details as well as the algorithms used in each of them.



## Chapter 2

# Multicast Routing

An introduction to combinatorial algorithms for unicast routing was presented in the previous chapter. In that context, routing was defined as the process of sending packets from an originating node to a single destination. While this is the most common option for network routing protocols and applications, it is not the only relevant scenario for packet routing.

An equally important routing scenario occurs whenever more than two nodes need to exchange data. When this happens, handling of the routing process requires the creation of a logical link among the involved elements, comprising a set of required nodes. The resulting routing tree needs to satisfy some application-specific constraints to be considered as a valid solution for the problem.

A network route that connects more than two nodes at the same time is referred to as a *multicast route*. Networks that employ this type of routing strategy are classified as *multicast networks* [144].

Multicast routing is mainly applicable to networking scenarios where data needs to be shared by a group of users. Such groups can range from relatively small to a possibly large number of clients.

Using common unicast protocols to perform the described logical connection between elements of a group of user nodes may be inefficient. Network resources (mainly bandwidth) may be wasted if copies of identical data are sent individually to each required node using the same network links. Such an arrangement can ultimately waste valuable bandwidth and equipment time, which would be better used to increase the quality of service to users.

While wireless systems can partially avoid this type of waste (since they use broadcasting to connect nodes that are located inside their range of wireless connectivity), this is not an option for wired networks. In such a scenario, alternative solutions that optimize the use of network resources must be investigated.

Our focus in this chapter is on the optimization of resource usage in multicast networks in general. We start by describing what type of resources can be managed in a multicast network. We give an introduction to some of the optimization problems occurring in this area, along with a sample of techniques that have been developed for their solution.

We are particularly interested in problems that are computationally difficult to solve (NP-hard). Some of these problems may only be solvable with the help of approximation and heuristic methods capable of finding near optimal solutions.

## 2.1 Introduction

We start our discussion with a simple example of multicast routing application. Consider a company that needs to update the software that is used across several offices located in different areas of the United States. For example, the company headquarters may be located in Chicago, while the subsidiary offices are in several cities of the east and west coast of USA. The company has a large amount of data that needs to be updated periodically, and links in the network are priced according to bandwidth usage. Initially, the objective of the company is to reduce the amount of bandwidth used in this operation. Figure 2.1 shows the configuration of such a network.

When faced with a network engineering problem such as this one, the simplest approach may be to just open a different connection to each site, until it can be guaranteed that all client locations are serviced. The advantages of doing this include the simplicity of the solution, since we do not require any additional computational effort. The resulting algorithm runs in polynomial time, and finding the routes will not present a great burden in terms of computing effort.

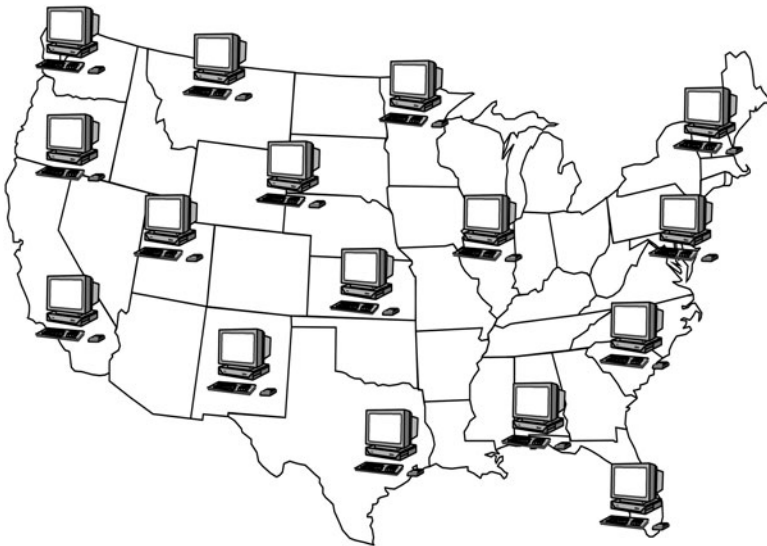


Fig. 2.1 Simple instance of multicast routing problem

The greatest disadvantage of this method, however, is that bandwidth is being almost certainly wasted, due to the repetitive nature of the transmissions that must be accomplished over the same set of links. Anytime a link is used to connect more than one client, it is carrying the same data more than once. Bandwidth could be reduced if, instead of sending data over the same link, we could just make copies of it whenever necessary.

This observation is the basis of *multicast routing*. Data is sent to nodes only once, and multiple copies are made as necessary until all destination nodes are satisfied, each node receiving a single copy of the original data. The replication points are usually known as or *multicast retransmission nodes*, or simply *cache nodes*.

### 2.1.1 Definitions

A *multicast network* is a set of nodes and connections, where data can be sent using a multicast strategy. A multicast network operates by interconnecting a source node to a set of destinations.

A *multicast session* is the collection of transmissions that need to be supported by the multicast network.

A *multicast group* is the name given to the set composed by the source and destination nodes that need to be connected in a multicast session.

A multicast group can be classified as static or dynamic, depending on how the members of the group can be allowed to change. A static group is one where the memberships cannot change for the entire live of the group. Starting with Wall [187], the problem of routing information in static groups is frequently modeled as a type of Steiner tree problem.

A more flexible organization for multicast groups occurs when members are allowed to enter or leave the group as necessary. Such multicast groups are called dynamic. Dynamic groups require extra care, as insertion and deletion policies must be defined by the underlying protocol (see [193]). One of the main challenges on dynamic groups is to maintain the quality of the routing tree without expending excessive time on each node addition or deletion.

Multicast groups can also be classified according to the relative number of users, as described by Deering and Cheriton [50]. In *sparse groups*, the number of participants is small compared to the number of nodes in the network. On the other hand, when most of the nodes in the network are engaged in multicast communication, the groups involved are called *pervasive groups* [186].

A *multicast tree* is a set of arcs without loops that connects the nodes in a multicast group. Generally, multicast trees are the preferred way of generating routes in multicast networks, since their use avoids redundancy in data transmission. Most algorithms studied in this book will assume that a multicast tree is the required solution for the multicast routing problem.

### 2.1.2 Applications

Applications of multicast routing have a wide spectrum from business to government and entertainment. One of the first applications of multicast routing was audio broadcasting over IP networks. In fact, the first real use of the Internet MBONE (Multimedia Backbone, created in 1992) was to broadcast audio from Internet engineering task force (IETF) meetings over the Internet [63].

A prime example of multicast application is financial data delivery, where one has a small number of data sources that are subscribed by millions of users. Trading data from financial centers such as Wall Street is requested by a huge number of interested users, ranging from investors to regulators, media and financial web sites.

Multimedia distribution systems can benefit directly from improved multicast technologies [155]. There is a huge amount of video, music, and related entertainment content that needs to be delivered to users worldwide. A multicast system can be used to effectively connect the consumers of such data to content providers.

Software delivery and update is another interesting application of multicast, as described at the beginning of this chapter. The amount of software installed in certain corporations is huge, and would greatly benefit from a multicast infrastructure for improved delivery.

Multicast networks can also be used in several other applications, including business-to-business (B2B) solutions, management of large projects, groupware [37], and even game communities [153].

A common feature of all these applications is that the data can be accessed over the Internet; therefore, instances of the networks considered can increase quickly. Adding this to the fact that most such problems are already NP-hard shows the difficulty of finding optimal solutions for problems arising in multicast routing.

### 2.1.3 Optimization Objectives

Given a network, a source node, and a set of destinations, the problem of routing on a multicast network can be quantified using several objectives, or metrics.

- *Delay Minimization:* A common objective is to minimize the delay of the routes used. For example, in video conferencing applications, it is important to limit the maximum delay experienced by users. This is called the minimum delay multicast routing problem.

If minimizing delay is the only goal in the problem, it can be solved with a polynomial time algorithm. This is possible because, for delay considerations, the destinations can be viewed as independent of each other. If the minimum delay path is calculated for each destination, then the maximum of these values can be taken as the solution for minimum delay routing. The repeated application of a shortest path algorithm such as, for example, the Dijkstra's algorithm [55], can be used to achieve this objective.

Delay optimization is frequently associated with Quality of Service (QoS) measures [107]. In some applications, it is important to maintain high levels of network service quality, and the QoS criteria are mostly related to data delivery time. The best example of application that needs this quality of service is video-conferencing.

- *Cost Minimization*: A common objective is simply to minimize the costs associated with the connections used to route packets, that is, minimizing the total cost of the routing tree. The resulting problem is known as minimum cost multicast routing. This problem also employs an additive metric but is considerably harder to solve than the delay minimization problem. In this case, the optimization goal can be proved to be equivalent to the *minimum Steiner tree*, a classical NP-hard problem [76].
- *Congestion Minimization*: Another example of optimization goal is to minimize the maximum network congestion. The congestion on a link is defined as the difference between capacity and usage. The higher the congestion, the more difficult it is to handle failures in some other links of the network. Also, higher congestion makes it harder to include new elements in an existing multicast group, and therefore is an undesirable situation in dynamic multicast. Thus, in a well designed network it is interesting to keep congestion at a minimum.
- *Cache Placement*: Another metric for multicast routing that is explored in this book is the determination of the optimum location of cache nodes. In this problem, the goal becomes to use intermediate nodes (transmitters) to satisfy capacity requirements [144]. At the same time, it is necessary to supply all client nodes with data from the source. A number of variations of this problem exist, most of them being NP-hard.

The optimization objectives described above can be optimized separately as a single objectives, or they can be combined as a multiple objective problem. Moreover, some applications define problems where a single objective is determined, and other objectives may appear as side constraints. For example, a version of the multicast routing problem may require the determination of a minimum cost routing tree, where the delay is bounded to a fixed value. Such definitions of the problem are very common in practice, depending on the desired goal of the application.

## 2.2 Multicast Technologies

The idea of sending data to a large number of users is common in systems that employ broadcasting. Radio and TV are two standard examples of broadcasting systems which are widely used. On the other hand, networks were initially designed to be used as an end-to-end communication systems, more similar to the way telephony services work.

The TCP/IP protocol stack, which is the main technology underlying the Internet, uses routing protocols for delivery of packets for single destinations. Most of these

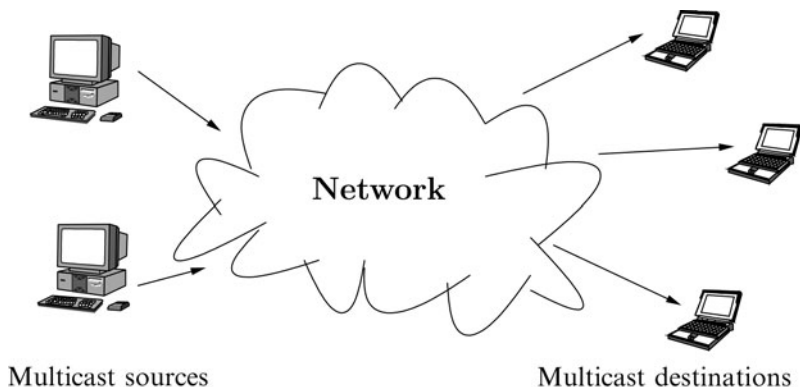


**Fig. 2.2** Conceptual organization of a multicast group

protocols are based on the calculation of shortest paths. A good example of a widely used routing protocol is Open Shortest Path First (OSPF, see [134, 183]), which is used to compute routing tables for routers inside a subnetwork.

In OSPF, each router in the network is responsible for maintaining a table of paths for reachable destinations. This table can be created using Dijkstra's algorithm [55] to calculate shortest paths from the current node to all other destinations in the current sub-network. This process can be done deterministically in polynomial time, using at most  $\mathcal{O}(n^3)$  iterations, where  $n$  is the number of nodes involved.

However, with the advance of the Internet and other networked services, the necessity appeared for protocols able to target larger audiences. This trend became more important due to the development of new technologies such as virtual conference [165], video on demand [202], and group-ware [61]. This series of developments gave momentum for the proposal of multicast routing protocols (Fig. 2.2).

Dalal and Metcalfe [45] were among the first to propose nontrivial algorithms for routing of packets in a multicast network. From then on, many proposals have been made to create the technology supporting multicast routing, such as by Deering [48], Eriksson [63], and Wall [187].

To handle the requirements of multicast routing, many proposals of multicast technologies have been presented in the last decade. Some examples of the array of multicast protocols and technologies include the following: PIM – Protocol Independent Multicast [52], DVMRP – Distance-Vector Multicast Routing Protocol [50, 186], MOSPF – Multicast OSPF [135], and CBT – Core Based Trees [14].

In the next sections, we discuss some of the major technologies supporting multicast networks. These are important not only as practical technologies but also as a display of implementation options for multicast routing.

## 2.3 Major Multicast Implementations

Below, we list some of the most important protocols and a few facts about how they are used to provide multicast connection for multiple nodes:

- *Protocol Independent Multicast* – Protocol Independent Multicast (PIM) is a general protocol that is able to connect nodes using different underlying routing technologies. PIM can be further divided into two main versions: Sparse mode PIM (PIM-SM [49]) and dense mode PIM (PIM-DM [53]). Sparse PIM is more adapted to the case where there is a small number of source/destination nodes.
- *MBONE, the Multicast Backbone* – MBONE is a protocol that works as an overlay network over standard IP. Therefore, it allows multicast to be provided even in the absence of a previous network infrastructure to support multicast protocols. MBONE has been very successful in allowing quick deployment of multicast over the Internet. The protocol was officially introduced through the transmission of a session of the Internet Engineering Task Force in 1992. A large number of networks have been added to the original MBONE, but nowadays, due to lack of nonacademic support, it is not viewed as a generally available method for multicast deployment.
- *Distance Vector Multicast Routing Protocol* – The DVMRP protocol has been developed as an extension of the routing information protocol (RIP), a widely used unicast protocol, which is one of the basic protocols used in the Internet. RIP has been traditionally used to distribute routing information inside the borders or large networks.

DVMRP can be described as a source-based protocol, since it computes the routes starting from the source up to the set of destinations. A more complex version of DVMRP is the *hierarchical DVMRP*, which divides the network into several lower level domains. The problem is then solved in each of the domains, and the resulting solution combined to form a global routing tree.

- *Multicast Open Shortest Path First* – In the same way that DVMRP tries to extend RIP to handle multicast, the MOSPF protocol is a multicast extension of the well-known OSPF protocol for unicast routing. OSPF is another protocol used for routing inside IP networks, which is based on the use of link-state algorithms. The OSPF protocol uses shortest path algorithms (such as the Dijkstra algorithm) to determine the path to be followed by each packet in the network.

The MOSPF protocol extends the OSPF architecture by adding the notion of sub-domains. It first divides the network into several sub-domains. Then, the algorithm computes a backbone connecting the domains, in an attempt to reduce the complexity of the problem. MOSPF is in active development, and is described by Moy [135].

For readers interested in the internal architecture of the multicast systems cited above, a more comprehensive discussion is given by Wittmann and Zitterbart [198]. Also see Levine and Garcia-Luna-Aceves [119] for a detailed comparison of these technologies.

## 2.4 Basic Routing Techniques

In this section, we discuss in more detail some of the basic combinatorial problems occurring in the implementation of multicast networks, along with routing algorithms that have been proposed to solve these issues.

We start by an introduction to terminology used for the mathematical modeling of the problems. Much of this terminology is based on graph theory concepts.

In the sequence, we discuss some basic combinatorial problems that are frequently seen in connection with multicast routing. Such problems have played an important role in the development of practical algorithms for multicast, as they are frequently referred to in the literature of the field.

### 2.4.1 Graph Theory Terminology

We use a few concepts from graph theory, and the terminology discussed in the next paragraphs will be used throughout the book.

A graph is a mathematical object composed of a set of nodes and a set of node pairs. A graph is denoted by  $G = (V, E)$ , where  $V$  is the set of nodes and  $E$  the set of edges. In our applications, graph nodes represent network elements in the network, such as routers, sources, or destinations. On the other hand, edges represent network links.

Graphs in this book are usually considered to be undirected and without loops. That is, the edge  $(i, j)$  is equivalent to  $(j, i)$ , and there is no edge  $(i, i) \in E$ , for  $i \in V$ . When directed graphs are needed, this will be explicitly stated. We use  $N(v)$  to denote the set of neighbors of a node  $v \in V$ . Also, we denote by  $\delta(V)$  the number of such neighbors.

With each edge  $(i, j) \in E$ , we can associate functions representing characteristics of the network links. The most widely used of such functions are capacity  $c(i, j)$ , cost  $w(i, j)$ , and delay  $d(i, j)$ , for  $i, j \in V$ .

- *Capacity function:* For each edge  $(i, j) \in E$ , the associated capacity  $c(i, j)$  represents the maximum amount of data that can be sent between nodes  $i$  and  $j$ . In multicast applications, this is generally given by an integer multiple of some unity of transmission capacity, so we can say that  $c(i, j) \in \mathbb{Z}_+$ , for all  $(i, j) \in E$ .
- *Cost function:* The function  $w(i, j) : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{R}$  is used to model any costs incurred by the use of the network link between nodes  $i$  and  $j$ . This includes leasing costs, maintenance costs, etc.
- *Delay function:* Some applications, such as multimedia delivery, are sensitive to transmission delays and require that the total time between delivery and arrival of a data packet be restricted to some particular maximum value [69]. The delay function  $d(i, j)$  is used to model this kind of constraint. The delay  $d(i, j)$  represents the time needed to transmit information between nodes  $i$  and  $j$ .

As a typical example, video-on-demand applications may have specific requirements concerning the transmission time. Each packet  $i$  can be marked with the maximum delay  $d_i$  that can be tolerated for its transmission. In this case, the routers must consider only paths where the total delay is at most  $d_i$ .

This is a short introduction to graph theory terminology used in this and the next chapters. For a more thorough overview of graph theory, one can check introductory books such as Diestel [54].

### 2.4.2 Flooding Algorithm

The flooding algorithm, described in some detail in the previous chapter, can also be successfully employed in the context of multicast routing. The main observation that supports this use is noticing that packets, once accepted by a node, are resent to all of its neighbors. Therefore, all destinations for a packet can be found using the flooding algorithm – even if there is more than one destination, as is the case with multicast routing. Therefore, making only a few changes in the major strategy, one can use the flooding algorithm of Sect. 1.2 for multicast routing.

Although being simple to implement, the flooding method has the same main disadvantage here as in the unicast case: it uses too much resources of the network. For example, bandwidth is wasted since a packet is sent over all available links. Moreover, at each node, the router needs to store information about the packets received, which increases both memory consumption and processing time. Since packets must be retained in memory for a long time, this makes it difficult to scale the use of flooding for more than a few nodes.

Despite these problems, several methodologies have been proposed to tackle the issues raised by flooding. For example, the reverse path-forwarding algorithm, proposed by Dalal and Metcalfe [45], is a method used to reduce the usage of network resources associated with the flooding algorithm. The idea is that, for each node  $v$  and source node  $s$  in the network,  $v$  will determine in a distributed way what is the edge  $e = (u, v)$ , for some  $u \in V$ , which is in the shortest path from  $s$  to  $v$ . This edge is called the parent link.

The parent link can be determined in different ways, and a very simple method is the following one: select  $e = (u, v)$  to be the parent link for source  $s$  if this was the first edge from which a packet originating from  $s$  was received. With this information, a node can selectively drop incoming packets, based on its source.

If, during the routing phase, a packet  $p$  is received from a link which is not considered to be in the shortest path between the source node and the current node, then  $p$  is discarded. Otherwise, the node broadcasts  $p$  to all other adjacent links, just as in the flooding algorithm. The parent link can also be updated depending on the information received from other nodes.

There are still other strategies that have been proposed to improve flooding for multicast routing. For more information on these and other improvement schemes see, for example, [171].

### 2.4.3 Algorithm Classification

During the last few years, a number of basic techniques were proposed for the construction of multicast routes. Diot et al. [56] identified and classified some of the most common techniques used in the literature for creation and optimization of routing trees. They described these techniques as being divided into source-based routing, Steiner tree-based algorithms, and center-based tree algorithms.

In *source-based routing*, a routing tree rooted at the source node is created for each multicast group. This technique is used, for example, in the DVMRP and PIM protocols. Some implementations of source-based routing make use of the reverse path-forwarding algorithm, presented in Dalal and Metcalfe [45].

Source-based routing has been used in several algorithms. It is effective, but there are some issues that with the quality of solutions found using this algorithm. For example, Sriram et al. [175] observed that this technique does a poor job of routing small multicast groups, since it tries to optimize the routing tree without considering other potential users not in the current group.

Among the source-based routing algorithms, the *Steiner tree-based methods* focus on minimization of tree cost. This is probably the most used approach, since it can leverage the large number of existing algorithms and heuristics for the Steiner tree problem. There are many examples of this technique, and a few of them can be found in Bharath-Kumar and Jaffe [24], Wall [188], Waxman [193], and Wi and Choi [195].

In contrast to source-based routing, *center-based tree algorithms* create routing trees with a specified root node. This root node is computed to have some special properties, such as, for example, being closest to all other nodes in the routing group.

Center-based methods are well suited to the construction of shared trees, since the root node can have properties that are interesting to all multicast groups using the network. For example, if the root node  $v$  is the topological center of a set of nodes, then by definition,  $v$  is the node which is closest to all members of the involved multicast groups.

In the particular case of the topological center algorithm, the problem of finding the root node becomes NP-hard. However, there are other versions of the problem which are much easier to solve. An important practical application of this idea occurs in the implementation of core-based tree (CBT) [14].

Still another interesting method for data distribution in multicast groups, which is not included in the classification discussed above, is called ring based routing [12, 141]. The main idea of this method is to have a ring linking nodes that participate in a group. The reason for using a ring as the connecting structure is to explore their properties to minimize cost and improve reliability. Notice, for example, that trees can be broken by the failure of just one link; on the other hand, rings are 2-connected structures, which offer a more reliable interconnection.

### 2.4.4 *Shortest Path Problems with Delay Constraints*

Given a graph  $G(V, E)$ , a source node  $s$  and a destination node  $t$ , with  $s, t \in V$ , the shortest path problem consists of finding a path from  $s$  to  $t$  with minimum cost. The solution of single-source/single-destination shortest path problems is a required sub-step in most implementations of routing algorithms. As we mentioned in the previous chapter, this problem can be solved in polynomial time using standard algorithms, such as in [22, 55, 71, 104].

However, other variations of the shortest problem are harder, and cannot be solved exactly in polynomial time. An example of this occurs when we add delay constraints to the original problem. The delay constraints (also referred to as quality of service, or QoS constraints) require that the sum of the delays from the source to each destination be bounded by some threshold. When a bounded delay is introduced, the shortest path problem becomes NP-hard, as described in [76].

When an efficient solution is necessary, the best approach is to use a heuristic algorithm. Several heuristics have been proposed for the shortest path problem with delay constraints. A few examples of good heuristics for these problems can be found in [50, 179]. A more general discussion of such heuristics is given by Salama et al. [167].

Some algorithms for shortest path construction are more appropriate for distributed implementation, while others have features that make them harder to use in a distributed scenario. As mentioned by Cheng et al. [35], a disadvantage of the distributed Bellman–Ford algorithm for shortest path computation is that is difficult to recover from particular events such as link failures, from the bouncing effect caused by loops [174], and from termination problems caused by disconnected segments.

A chief requirement for shortest path algorithms used in multicast routing is to have a scalable distributed implementation. The problems associated with the requirements for distributed implementations of shortest path algorithms are discussed by Cheng et al. [35]. That paper also proposed some techniques aimed at overcoming these limitations.

### 2.4.5 *Delay Constrained Minimum Spanning Trees*

In the minimum spanning tree (MST) problem, given a graph  $G(V, E)$ , we need to find a minimum cost tree connecting all nodes in  $V$ . This problem can be solved in polynomial time by Kruskal's algorithm [115] or Prim's algorithm [157]. However, similar to the shortest path problem, the MST problem becomes NP-hard when delay constraints are applied to the required paths in the routing tree. This fact can be easily proved, since the minimum spanning tree problem is a generalization of the minimum cost path problem.

Due to the computational complexity of the delay constrained MST, several heuristic algorithms have been devised for its solution. In the remaining of this section, we describe an approximate solution method for the delay constrained minimum spanning tree problem, proposed by Salama et al. [168]. The algorithm, named Bounded Delay Broadcast (BDB), is a simple heuristic that provides near optimal solutions. The BDB heuristic has the objective of running quickly and generating fast solutions for commonly found instances of the delay constrained multicast routing problem.

The BDB algorithm proceeds as follows. In its first phase, the BDB algorithm tries to incorporate links, ordered according to increasing cost, but without creating cycles. The technique used is similar to Prim's algorithm, in which it incrementally adds elements to the solution, stopping only when there is a complete tree that satisfies the requirements of the problem. In particular, at each step of the BDB algorithm, the goal is to ensure that the current (partial) solution satisfies the delay constraints for multicast routing.

If this requirement is not satisfied by the current partial solution, then a relaxation step is carried on, which consists of the following procedure. If a node can be linked by an alternative path, while reducing the delay, then the new path is selected. If, after this relaxation step, there is still no path with a suitable delay for some node, then the algorithm fails and returns only a partial answer.

Other examples of algorithms for computing delay constrained spanning trees include, for example, Chow [38]. In that paper, such an algorithm is used in the context of finding new routing tree, based on existing routes. The result of the delay constrained MST is then used to combine different routes into one single routing tree.

For additional information on delay constrained routing, one can refer to Salama et al. [169], where a detailed comparison of diverse algorithms for this problem is presented.

### 2.4.6 *Center-Based Trees and Topological Center*

Some routing technologies, such as PIM and CBT, employ a technique known as center-based trees [166] to improve the quality of the routing used by the protocol. Many of the techniques used in such center-based technologies were presented in [188].

In the CBT approach for multicast routing, the first step consists of finding the node  $v$  that is the topological center of the set of senders and receivers. A topological center for a given graph  $G(V, E)$  is defined as the node  $v \in V$  which is closest to any other node in the network, that is, the node  $v$  which minimizes  $\max_{u \in V} d(v, u)$ .

Once a topological center node has been identified for a set of source and destination nodes, it is possible to construct a routing tree rooted at  $v$ . Such a tree has some interesting properties because it can be more easily manipulated by adding and removing elements as necessary.

The basic reasoning behind center-based trees is that the topological center is a better starting point for creation of a routing tree. The topological center, due to its position, is expected to change less than other parts of the tree. Therefore, rooting the tree on the topological center makes it more stable relative to changes that might occur in the membership of the multicast group.

The CBT scheme provides a clear departure from the idea of rooting the routing tree at the sender, which is the most common way of creating routing trees. Due to the properties of the topological center, a center-based tree can be more easily extended to be used by multiple multicast group at the same time.

Finding a topological center is, however, a NP-hard problem [14]. As a result, some common approaches involve the use of root nodes that are not exactly the topological center, but which can be thought of as a good approximation of it. Along these lines, we have some algorithms proposed in the literature using alternatives such as core points [14] and rendezvous points [51].

It is interesting to note that, for simplicity, most of the papers which try to create routing trees using center-based techniques simply disregard the original NP-complete problem and try to find approximations that are easier to calculate. It is not completely understood how good these approximations can be for practical instances; however, on the more practical side, Calvert et al. [29] gave an informative comparison of the different methods of choosing the center for a routing tree, based on experimental data.

## 2.5 Additional Restrictions

While the problems described above form the core of important questions in the area of multicast routing, there are other less common restrictions that play a role in the development of algorithms for multicast.

We discuss briefly the problems of using symmetric links and the determination of routes in the presence of delay constraints.

### 2.5.1 *Nonsymmetric Links*

An interesting feature of real networks is that links are asymmetric. While this fact is rarely considered on research papers, it plays an important role in the design of new networks. In particular, the capacity of links in one direction is frequently different from the capacity in the opposite direction, due to features of networks that include congestion, limitations of the transmission medium, as well as requirements imposed by management.

A few algorithms for this scenario have been proposed in the literature. In one of these algorithms [161], the minimum cost routing tree was modeled as a minimum Steiner tree problem with symmetry constraints. In this way, it is possible to pass the asymmetric restriction from the network itself to the Steiner tree model.

The algorithm proposed in Ramanathan [161] provides an approximation guarantee for the problem, with constant worst case performance. The resulting algorithm has also the nice characteristic of being parameterizable, which means that it can trade execution time for accuracy when necessary.

### 2.5.2 Delay Variation

Another characteristic of real networks that is normally disregarded in research papers is the occurrence of delay variation. *Delay variation* is defined as the difference between the minimum and maximum delay incurred by a specific routing tree.

In some applications, it is interesting that delay variation be limited to a specific range. That is, it may be interesting to define a limit on difference between arrival times for information at specific clients. For example, of practical use, consider a delivery network for video, where it can be desirable that all nodes receive the same information at about the same time.

A few researchers have targeted the construction of routing trees where degree variation is important. For a deeper treatment of this issue, one can refer to Rouskas and Baldine [164], where an optimization approach has been proposed to minimize delay variations.

## 2.6 Additional Resources

Routing for multicast systems is an area where new ideas continue to appear frequently. Due to space limitations, most of them could not be included in this book. However, in the next paragraphs, we provide references to some of the most exciting new trends in this evolving area. Most of these topics deserve additional research and hold promise for improved applications.

- *Stress testing*: One of the challenges with multicast systems is how to determine the appropriate behavior in the face of failures and unusual conditions. A possible method used to determine such dynamic behavior is the application of stress testing. Such methodology has been discussed, for example, in Helmy and Gupta [89].
- *Blocking*: Blocking happens when transmissions from multiple sources need to use the same links. Contention for the use of resources in multicast systems may eventually result in blocking for some of the participants in the multicast group. The issues related to blocking on multicast networks has been investigated by Hwang [93], where a multicast 3-stage method is used to prevent blocking. In that paper, it has also been determined the necessary and sufficient conditions for the existence of nonblocking multicast networks.

- *Dual-homing*: Dual-homing is an implementation scheme where each node can possibly participate in more than one multicast tree. In this situation, it is interesting to determine a best routing policy that provides protection under failures of the underlying system. As an example of approach for solving this problem, one can refer to a paper by Yang et al. [201]. In that paper, the authors discuss a method for creating what they refer to as a *protection tree*, based on such dual-homing architectures.
- *Security of Multicast Systems*: Security and identification is also a major issue in networks, specially when multicast is involved. Recent proposals have tackled these issues using key management methods. In Eltoweissy et al. [62], the authors provide a combinatorial optimization algorithm for group key management. A related method proposed by Zhu [205] uses optimization techniques to improve the routing tree structure to allow for secure key management on such systems.
- *Economic Analysis*: Operations research has always been concerned with the economic use of scarce resources. On multicast networks, a number of resources are shared among users of the service. As an example of possible approaches in this area, an economic analysis of multicast is provided by Wang et al. [191]. Among their results, it is provided an analysis of the fair sharing of payments by members of one or more multicast groups.
- *Switching Networks*: Another work in multicast routing that seems promising is the design of multicast trees in switching networks [200]. Here, the issue of blocking is also important, and algorithms need to be designed to take advantage of the special topology of switching networks.

For more information about multicast networks in general, one can consult the surveys by A.J. Frank [5] and Paul and Raghavan [156]. A detailed description of current implementations of multicast over IP networks is given on the Internet Draft by Semeria and Maufer [171], which is available online. Other related literature includes Du and Pardalos [58], Pardalos and Du [152], Wan et al. [189], Pardalos et al. [150, 151], and Pardalos and Khoury [148, 149].



## Chapter 3

# Steiner Trees and Multicast

Steiner trees have for many years been an important problem in Operations Research. They have been used to model a variety of applications, ranging from transportation to computer aided design. As such, Steiner trees have received a lot of attention in the optimization literature, with several theoretical and practical results appearing each year.

Steiner trees also have an important role on the design of algorithms for multicast routing. Since multicast requires the connection of routes from a source to multiple destinations, it can be naturally modeled as a version of Steiner tree problem on graphs.

While this model provides useful features for the analysis of algorithms for multicast routing, it also presents a number of problems. First, the Steiner tree problem is NP-hard in general. Therefore, any algorithm that uses such a model has to find ways to cope with the great complexity of solving the problem exactly. Second, the Steiner tree model does not capture many of the intrinsic features of multicast routing, which range from topological issues to practical considerations such as delay and contention problems.

In this chapter, we examine the relationship between computing multicast routing trees and solving the Steiner tree problem. We start with a quick review of the Steiner tree problem and the techniques available for its solution. We then discuss algorithms for multicast routing that can be considered as extensions of existing algorithms for Steiner tree.

### 3.1 Basic Concepts

The traditional problem of routing consists of finding a path between two nodes in a network. In particular, existing shortest path algorithms may be used to compute such paths. Despite the large number of feasible solutions (since in general, there

is an exponential number of paths connecting two nodes in a graph), this problem is known to be solvable in polynomial time using combinatorial algorithms such as Dijkstra's.

In multicast routing systems, on the other hand, we wish to find a subset of network arcs linking the source node to the set of destinations. Additionally, we allow nodes in the network to be used as replication points. That is, data packets arriving to a node can be simultaneously sent to several destinations.

This situation is in some ways similar to the Steiner tree problem on graphs. In the Steiner problem, we are given a graph  $G = (V, E)$  and a set  $R$  of required nodes. The objective of the problem is to connect the required nodes  $R$  using the remaining nodes of the graph ( $V \setminus R$ , which are known as Steiner points) as intermediate nodes if necessary. Assuming that each edge in the graph has an associated cost, the goal of the problem is finding such a solution with minimum cost. The cost for the solution is given by the sum of costs of edges included in the solution set.

The solution for the Steiner tree problem is a tree, since it is minimally connected – no edge in the solution can be removed without disconnecting its components. If we denote a solution to the Steiner tree problem by  $T$ , with each edge  $e \in E$  having a cost  $c_e$ , then the objective function can be represented as

$$\min c(T) = \sum_{e \in T} c_e.$$

## 3.2 Steiner Trees on Graphs

In the Steiner tree problem, the goal is to find a minimum cost tree that connects the required nodes. Nonrequired nodes are also called Steiner nodes, that is, nodes that can be used only as necessary by the connecting tree.

The Steiner tree problem was initially proposed as a problem in recreational mathematics on a geometrical setting. The lines connecting the required points were supposed to be on the general plane. The problem eventually became more important when applied to graphs, where the connecting points are defined by the set of nodes.

When applied to graphs, the Steiner problem becomes a discrete optimization problem, since only a finite number of connecting trees exist. To find the optimal solution, one needs to execute a combinatorial optimization algorithm that can find the best solutions among a finite but large number of possibilities.

Due to its occurrence in practical applications and its usefulness in the derivation of theoretical results, the Steiner tree problem on graphs has become one of the most important problems in combinatorial optimization.

The Steiner tree problem on graphs can be considered a classical NP-hard optimization problem [76], and has a vast literature on its own [19, 59, 60, 94, 95, 114, 181, 196, 197]. In this section, we discuss only some of the most useful algorithmic results. For additional information about this problem, one can consult the surveys by Winter [196], Hwang and Richards [94], Hwang et al. [95].

### 3.3 Modeling Multicast Routing Constraints

The Steiner tree technique for multicast routing consists of using the Steiner problem as an initial model for the construction of a multicast tree. In general, when applying this strategy, it is considered that there is just one source node for the whole multicast group. The set of required nodes is then defined as the union of source and destinations.

Using the Steiner tree as a model for multicast routing construction is one of the most common techniques used in the literature. Several papers either modify the details of algorithms for the Steiner tree problem, or use their results as the initial step for the creation of a multicast routing tree. For example, see [18, 32, 38, 91, 109–112, 161].

Most often, however, routing problems that need to be solved in real life multicast networks have additional constraints that are not present in the formulation of the Steiner tree problem. These are generally related to technological requirements that need to be satisfied. For instance, many routing algorithms need to consider quality of service (QoS) as part of the problem formulation. An example where such real-world requirements become a necessity occurs in video-conference applications.

When solving the multicast routing problem with QoS constraints, it is necessary to find a solution where no path from  $s$  to  $d \in D$  takes more than  $\Delta$  units of time. In this case, the  $\Delta$  parameter defines the quality of the overall network.

Another important example of additional constraints that can be added to a Steiner tree model is guaranteeing reliability properties in a network. Some of these applications require a safety margin of use for any network link. In such a case, no network link can use more than  $\beta\%$  of its total capacity. The objective of such constraints is to allow for reserve capacity on the network, which may be required by contract, or for security measures. A typical example is the use of spare capacity in the case of failures of other network links.

It is easy to show that the problem extensions discussed above are as hard as the original Steiner tree problem. For a particular example, the Steiner tree problem is equivalent to the multicast routing problem with quality of service constraints where  $\Delta = \infty$  and  $\beta = 100\%$ .

Considering the close relationship between these problems, one can devise simple algorithms where an approximate solution to Steiner tree can be used as a starting point for a solution of the whole problem. The general scheme consists in finding a solution to Steiner tree in reasonable time, then modifying the resulting solution as necessary to satisfy additional constraints.

In the remaining of this chapter, we discuss different versions of the Steiner tree problem, and how they can be used as a starting point to solve some of the problems arising in multicast routing. Algorithms for these variations of the Steiner tree problem, along with references, are also presented.

```

MST-Heuristic
1  Construct a complete graph  $K(R, E)$  where the set of nodes is  $R$ .
2  Let the distance  $d(i, j)$ ,  $i, j \in R$  be the shortest path from  $i$  to  $j$  in  $G$ .
3  Find a minimum spanning tree  $T$  of  $K$ .
4  Replace each edge  $(i, j)$  in  $T$  by the complete path from  $i$  to  $j$  in  $G$ .
5  Let the resulting graph be  $T'$ 
6  Compute a minimum spanning tree  $\hat{T}$  of  $T'$ .
7   $r \leftarrow true$ 
8  while  $r \neq false$  do
9     $r \leftarrow false$ 
10   if there is a leaf  $w \in \hat{T}$  which is not in  $R$  then
11     Remove  $w$  from  $\hat{T}$ 
12      $r \leftarrow true$ 
13   end
14 end

```

**Fig. 3.1** Minimum spanning tree heuristic for Steiner tree

### 3.3.1 KMB Algorithm

Since the Steiner tree is NP-hard, an initial trade off may be finding a solution that, while not optimal, has a theoretical guarantee of optimality. That is, given a threshold  $\epsilon$ , we may try to find an algorithm that always returns a solution within a factor  $\epsilon$  of the optimum for the given instance. This is the approach frequently used when creating algorithms with performance guarantee. Such methods are better known in the literature as *approximation algorithms*.

One of the best known heuristics for the Steiner tree problem provides such an approximation guarantee. It was proposed by Kou et al. [114], and frequently referred in the literature as the KMB heuristic. There is practical interest in this heuristic due to its simple implementation. It also provides a performance guarantee that is within a constant factor of the optimum for the problem. The steps of the KMB heuristic are summarized in Fig. 3.1 (Algorithm).

The KMB algorithm can be described as follows. Let  $K_n$  be the complete graph, created in such a way that  $n = |D \cup \{s\}|$ . We assume that the cost  $c_e$  of arc  $e = (v_1, v_2)$  in this complete graph (for all arcs  $e \in E(K_n)$ ) is equal to the minimum distance from  $v_1$  to  $v_2$  in  $G$ . The algorithm proceeds by finding a minimum spanning tree  $T$  on  $K_n$ . Then, the next step is to create a new graph  $G'$  with  $|V(G)|$  nodes. For each pair of nodes  $u$  and  $v$  in  $T$ , we add to  $G'$  a path composed by the same nodes that form the path from  $u$  to  $v$  in  $G$ . Finally, we find a tree in  $G'$  by simply removing all redundant edges in the usual way, that is, taking off the high cost edges whose removal will not disconnect the tree.

Although simple, this algorithm can be shown to provide a constant factor approximation guarantee, as formally stated below:

**Theorem 1.** *The KMB algorithm is a  $2 - 1/p$ -approximation algorithm for Steiner tree, where  $p$  is the number of required nodes.*

### 3.3.2 *The KMB Algorithm on Multicast Networks*

Due to its simplicity, the KMB algorithm has been directly applied to solve instances of the multicast routing problem. Wall [187] was the first to make a comprehensive comparison of how such a heuristic performs in problems occurring in real multicast networks.

For an example of the efficiency of the KMB algorithm, Doar and Leslie [57] reported that this heuristic can give results that are usually much better than the claimed guarantee, routinely achieving 5% of the optimal for a large number of realistic instances.

On the other hand, it is rarely enough to apply an algorithm such as KMB directly to a multicast routing problem. Although this might be possible, if the output is blindly applied using an unchanged algorithm, the resulting routes may not satisfy some of the technological constraints present on real world instances. However, we can slightly modify the results of the KMB algorithm and incorporate into it the necessary requirements of multicast routing. As a result, one can propose a general approach for the multicast routing problem (MRP) based on an approximate solution for the Steiner tree problem:

1. Run the KMB algorithm on a corresponding instance of the Steiner problem on graphs.
2. Take the solution from the previous step and modify it as necessary to satisfy routing constraints.
3. Repeat the above steps while the solution can be improved.

While this strategy can be very simplistic for more difficult variants of the MRP, it can also provide a solution that is usually good enough for our purposes. That is why this idea has been repeatedly investigated in the literature, as we will see throughout this chapter. The results of these implementations have been satisfactory to a number of applications, which shows the power of the KMB heuristic as a general method for solving problems related to Steiner tree.

### 3.3.3 *Other Algorithms for Steiner Tree*

While KMB is the most common heuristic for the Steiner tree problem, it is by no means the only one. Several other heuristics have been proposed in the literature.

Another basic heuristic for Steiner tree was proposed by Takahashi and Matsuyama [181]. This heuristic works in a way similar to the Dijkstra's and Prim's algorithms. The operation of the heuristic consists of starting with a tree composed of only one node. Then, the algorithm proceeds by increasing the initial solution tree along shortest paths (thus, it is classified as part of the broad class of path-distance heuristics).

Initially, the tree is composed of the source node only. Then, at each step, the heuristic searches for a still unconnected destination  $d$  that is closest to the current tree  $T$ , and adds to  $T$  the shortest path leading to the destination  $d$ . The algorithm stops when all required nodes have been added to the solution.

This algorithm has been used with success on a number of application, and it is particularly successful when there is ready availability to shortest path information. This is usually the case on routing applications, since shortest path computation is the basis for single-source single-destination routing protocols.

The use of path-distance heuristics has been popular among creators of multicast protocols, since the unicast routing information can be directly used in the calculation of the next edges to add to a solution to the Steiner tree problem.

A few examples of such ideas in practice include Bharath-Kumar and Jaffe [24], which present algorithms to simultaneously optimize the cost and delay of a routing trees. Also, on Waxman [193], we see a discussion of heuristics for cost minimization using Steiner tree algorithms that take into consideration the dynamics of inclusion and exclusion of members of multicast groups.

It is also important to note some of the limitations of the Steiner problem as a model for multicast routing. It has been pointed out by Sriram et al. [175] that Steiner tree techniques work best in situations where a virtual connection must be established. However, in the most general case of packet networks, such as the Internet, it does not make much sense to minimize the cost of a routing tree, since each packet can take a very different route. In this case, it is much more important to have distributed algorithms with low overhead.

Despite these limitations, Steiner trees are still useful as a starting point for more sophisticated algorithms for multicast.

### 3.4 Steiner Tree Problems with Delay Constraints

The most common way of applying the Steiner tree problem in multicast networks consists of minimizing the costs of edges in a tree representing communication costs incurred by the resulting routes. The objective then becomes minimizing the costs associated with the routing tree.

When this is the only problem that needs to be solved, it is possible to use the Steiner tree problem as a model, and apply any number of existing algorithms to the resulting instance of the problem, including some of the strategies discussed in the previous sections.

However, practical applications have additional requirements in terms of the maximum delay for delivery of the information. In a number of applications, it is important that the data be transferred within a given time limit. Such constraints can invalidate the result of solutions generated by simple Steiner tree implementations.

As a result, probably the most well studied version of the Steiner tree problem applied to multicast routing is the delay constrained version, as seen in papers such as Im et al. [96], Kompella et al. [109, 110, 112], Jia [99], Sriram et al. [175].

In the remaining of this section, we show some examples of methods used to give approximate solutions to the delay constrained version of Steiner tree.

### 3.4.1 *Adapting a Steiner Tree Algorithm*

As described in the previous section, one of the strategies used to solve the delay constrained Steiner tree problem is to adapt existing heuristics to cope with the addition of delay constraints.

As a first example of how this strategy can be used, we consider the heuristic proposed by Kompella et al. [112]. In this paper, we can see that a heuristic similar to the KMB algorithm [114] is employed as the initial step of the computation of the minimum cost routing tree.

The resulting algorithm is composed of three stages. The first stage consists of finding a closure graph of constrained shortest paths between all members of a multicast group. As in the KMB algorithm, the closure graph of  $G$  is a complete graph, which is composed of the set of nodes  $V(G)$  and, for each pair of nodes  $u, v \in V$ , and an edge  $(u, v)$  representing the cost of the shortest path between  $u$  and  $v$  for each  $u, v \in V$ .

In the second stage, Kompella's algorithm finds a constrained spanning tree of the closure graph. To do this, the heuristic uses a greedy method based on edge costs, which aims at finding a spanning tree with low cost. This step uses a heuristic algorithm to avoid the complexity of calculating the exact solution to the constrained spanning tree problem.

In the last stage of the algorithm, edges of the spanning tree found in the previous step are mapped back to paths in the original graph. At the same time, loops are removed using the shortest path algorithm, applied to the expanded constrained spanning tree.

The detailed complexity analysis of the proceeding algorithm, which can be seen in the original paper, shows that the whole procedure is  $\mathcal{O}(\Delta n^3)$ , where  $\Delta$  is the maximum delay allowed by the application.

Despite the similarity of the algorithm with the KMB heuristic, it should be noted that, unlike in the original algorithm, the resulting solution does not have any guarantee of approximation. This happens because, as a side-effect of adding delay constraints to the problem requirements, it becomes much harder to prove any guarantee of approximation. The same is true for many other versions of combinatorial optimization problems with additional constraints.

### 3.4.2 *Algorithms for Sparse Steiner Trees*

Another situation where we would like to compute good solutions to Steiner tree problem is the case when the network is sparse. This is a central question on telecommunications networks because it is well-known that such networks are sparse by design.

Having this property in mind, it is possible to devise algorithms that perform well when the network is sparse, that is, on networks where the number of links is approximately linear in terms of the number of nodes.

Sriram et al. [175] proposed an algorithm for constructing delay-constrained multicast trees which is optimized for sparse networks with static groups. This means that the whole configuration for the multicast network is available from the beginning, which may simplify the task of finding the best routing tree. The proposed algorithm is divided into two phases, as described in the following paragraphs.

The first phase of the algorithm is distributed, and works by creating delay constrained paths from source to each destination. In other words, paths are constructed in such a way that the total cost of each path is at most the value that was defined by the delay constraint.

In the algorithm, the paths are assumed to be created by a *unicast routing* algorithm. Therefore, the required information is already available as part of the data returned by the underlying unicast protocol, and does not need to be computed separately.

The second phase of the procedure uses the computed paths, as described above, to define a complete routing tree. To do this, each path found in the previous step is added sequentially to the routing tree. There is the possibility of cycle formation when new paths are added. Therefore, cycles are removed after each iteration until a valid partial solution is found.

The edge removal procedure works as follows. Suppose that on iteration  $i$  a new path  $P_i$  is added to an existing tree  $T_{i-1}$ . If this happens, each intersection  $P_i \cap T_{i-1}$  of the path with the old tree is then tested. By doing this, the algorithm is able to determine if only the part of  $P_i$  which does not intersect the existing solution can be used, while maintaining the validity of the delay constraints.

If this is possible, then the resulting tree becomes  $T_i$ , as a result of adding the nonintersecting part of the path. Otherwise, the algorithm employs a different procedure that is responsible for removing parts of the original tree in such a way that the formation of a cycle is avoided.

### 3.4.3 Combining Cost and Delay Optimization

Routing tree cost and delay are the most common optimization objectives for multicast routing. It is, therefore, not uncommon that one needs to optimize both measures to find a high quality routing tree for particular applications.

The literature has provided algorithms that aim at solving this class of instances of multicast routing problems. For example, such a heuristic for the delay constrained Steiner tree problem was presented by Feng and Yum [65]. The heuristic uses the idea of simultaneously constructing a minimum cost tree as well as a minimum delay tree, and then combining the resulting solutions in a single routing tree that could have good features of both.

Recall that a shortest delay tree can be created using the same algorithm used for shortest path computation. This can be done in polynomial time with the transmission delay being used as the cost function. This means that the hardest part of the algorithm consists in finding a minimum cost tree. Once both trees are available, the issue then becomes one of finding the best way to combine the two solutions.

In Feng and Yum [65], the algorithm used to compute the minimum cost/delay constrained tree is a modification of the well-known Dijkstra's algorithm. The new version of the classical algorithm will maintain each path within a specified delay constraint. To combine different trees, the algorithm employs a loop removal subroutine, which verifies if the resulting paths still satisfies the delay constraints.

The terms of computational complexity, this algorithm is also similar to Dijkstra's algorithm. Although it does not guarantee the optimality of the solution, or even provide a guarantee of performance for the results found, it is nonetheless a well performing algorithm able to find solutions for practical problems. Routing trees found using this method are made available after a short computation cycle, compared to other methods that aim at optimality.

### 3.4.4 Constrained Shortest Paths

Another possible method for computing multicast routing trees is to start from algorithms for computing constrained shortest paths over the underlying network. A constrained shortest path problem has a solution that is given by a shortest path. However, the total delay of the path (say,  $P$ ) is required to be bound by the sum of maximum delays in each of the edges  $e \in P$ .

This technique was used by [116], where two heuristics for the constrained minimum cost routing problem are presented. In the first heuristic, which is called "dynamic center-based heuristic," the goal is to find a center node. That will be the node to which all destinations will be linked using constrained minimum paths.

The center node  $c$  is initially calculated by finding a pair of nodes  $(u', v')$  with the property that the minimum path between them has highest minimum delay path. To calculate such a pair, it is necessary to find the shortest path distance  $d(u, v)$  between each pair of nodes in the graph, which can be easily done in  $O(n^3)$ . Once the distances are known, they can be ranked according to the delay incurred by using the shortest path between  $u$  and  $v$ , for all  $u, v \in V$ . One then can take one of the nodes in  $u, v$  as the center node.

Once the center node has been determined, one can start joining other destinations to the routing tree. This time, one can use minimum delay paths connecting the remaining destination to the existing tree. The process stops when all destinations have been assigned a path to the routing tree. Cycles are also avoided by removing edges with high associated delay.

The second heuristic proposed in Kumar et al. [116] is called "best effort residual delay heuristic." It follows a similar idea, but this time each node added to the

current routing tree  $T$  is considered to have a residual delay bound. New destinations are then linked to the tree through paths which have low cost and delay smaller than the residual delay of the connecting node  $v \in T$ .

### 3.4.5 *Link Capacity Constraints*

Many studies of multicast routing are concerned with versions of the problem occurring when additional constraints are involved. It is very common for telecommunications problems to incorporate constraints that are relevant only for the particular technology being used in the implementation. Such constraints may refer to factors such as link capacities or maximum distance allowed by existing network media, for example.

Jiang [100] discusses a version of the multicast routing problem that deals with link capacity constraints. In this case, a link  $e$  is supposed to carry at most  $c_e$  units of data. As a result, not every routing tree is feasible because it is possible that too many destinations are been supplied from the same link.

A solution for this optimization problem can be modeled after the Steiner tree problem with link capacity constraints. In Jiang [100] an application for this is mentioned, in which the multicast tree is used to implement videoconferencing applications. In such an application, user nodes may be required to act as sources during the establishment of the conference, in a way that is similar to file-sharing protocols.

One of the central ideas used in the proposed algorithm is that, since each user can become a new source, a distinct multicast tree can be created to support that user. This means that each user of the network can potentially spawn a separate routing tree that will serve new users. Some of the existing heuristics have been adapted to this situation, where routing trees need to be computed dynamically. Heuristics are necessary here due to the large effort necessary to find near optimal solutions when multiple routing trees are involved.

Computational results provided in Jiang [100] demonstrate that such heuristic methods are effective in defining new solutions for the multicast routing problem with link capacities, even when large size instances have been tested.

### 3.4.6 *Delay Bounded Paths*

If the paths between source and destinations are required to have delay given by a constant  $\Delta$ , then we say that the instance of the multicast routing problem has a delay bounded constraint. Such restrictions are used in applications that need low latency and fast response from users. For an example where such constraints are helpful, consider multicast networks where users need to interact, which is common on simulation applications and interactive games.

A solution for such a problem can be found by the application of heuristics, such as the one proposed by Zhu et al. [204]. The particular heuristic presented in that paper runs in time  $\mathcal{O}(k|V|^3 \log |V|)$ , where  $k$  represents the maximum allowed delay and  $|V|$  is the size of the underlying graph for the network being considered.

The algorithm can be divided into two distinct phases as follows:

1. In the first phase, a set of delay-bounded paths is constructed. That is, the goal is to create paths connecting the source  $s$  to each destination  $v \in R$ , where  $R$  is the set of required destinations. The algorithm also makes sure that the maximum allowed delay is respected. The set of paths generated in this way is then connected into a routing tree. At the end of this phase, the source node will be connected to all destinations in  $R$ . If there is a feasible solution, each connecting path will have a delay that has value at most equal to  $\Delta$ .
2. In the second phase, the algorithm tries to optimize this tree. The idea is, at each step, to reduce the total cost of the tree that has been calculated in the previous iteration. To do this, one can generate alternative paths connecting parts of the existing routing tree. If such alternate paths yield a better solution, they can be used to improve the current multicast routing tree.

Such an algorithm can also be useful to optimize objective functions other than total cost. For example, similar ideas can be used to minimize the maximum congestion in the network. That would be possible if changes were performed in the second phase of the algorithm to account for the differences in objective function.

In Zhu et al. [204], one can find experimental comparisons between the heuristic described above and the KMB heuristic for the Steiner tree problem [114]. The computational results have shown that the heuristic is able to find high quality solutions that are very close to what can be achieved by the KMB algorithm.

### 3.5 Sparsity and Delay Minimization

Most instances of the multicast routing problem share the sparsity property. This means that the number of links in the network is  $\mathcal{O}(n)$ , where  $n$  is the number of nodes in the graph, instead of  $\mathcal{O}(n^2)$ , which is the worst case for the number of links.

Sparsity is a convenient property to explore. The presence of sparsity can make some algorithms faster, since most network algorithms depend on the number of edges of the underlying graphs.

Chung et al. [39] proposed a heuristic that takes advantage of sparsity in the delay constrained minimum multicast routing. After considering the structure of such problems, the heuristic employs algorithms that are known to perform well when the involved graph instances are sparse. The output of such algorithms, then, can be used to find approximate solutions to the underlying Steiner problem. The resulting Steiner tree is then converted to a routing tree for the multicast problem.

In the heuristic discussed by Chung et al. [39], the KMB algorithm for Steiner tree is used to return two solutions: the first solution is calculated using the given link costs. The second solution is found by replacing the cost of each link by its associated delay. Using this approach, one has two solutions at the end of the process – each solution minimizing a different objective function.

The next step of the proposed algorithm is, therefore, producing a single solution that in some way optimizes the cost of both the routing tree as well as the maximum delay at the same time. To do this, the algorithm uses a method proposed by Blokh and Gutin [25], which is based on Lagrangian relaxation.

Using the Steiner tree algorithm to optimize both cost and delay is just one of the ways in which one can exploit the sparsity of multicast routing problems. In this particular case, though, a critique that can be made to [39] is that the goal of simultaneously optimizing the Steiner tree problem using both link costs and link delays is not really required in most applications.

For example, a solution can be optimal for delay according to the algorithm for Steiner tree. However, some paths from the source  $s$  to a destination  $d$  may still have delay greater than a given constant  $\Delta$ . This happens because the global optimum does not imply that each source–destination path is restricted to the maximum delay.

### 3.5.1 *Sparse Groups*

An important special case of multicast routing occurs when the number of sources and destinations is small compared to the whole network. This is the typical case for big instances, where just a few nodes will participate in a group at each moment. For this case, Sriram et al. [175] proposed a distributed algorithm which tries to explore the sparsity properties of the problem, which we describe in what follows.

The algorithm initially uses shortest paths as the starting point for the calculation of routes. In practice, this information is readily available through a unicast routing protocol. Therefore, one can easily leverage the existing path information provided by such unicast protocols.

Even when shortest paths can be made accessible through existing protocols, there are problems with using them directly in a routing tree. Adding such paths can inadvertently induce loops in the corresponding graph. Therefore, it is necessary to determine a method for loop removal to compute a routing tree.

In the algorithm presented in Sriram et al. [175], such intersections are treated and removed dynamically, as they appear. The algorithm starts this process by creating a list of destinations, ordered according to the cost of their delay constraints. Nodes with more stringent delay constraints have the opportunity of receiving new path assignments before other nodes. At the end, each destination  $d_i$  will independently have a path assignment connecting the destination  $d_i$  to the source  $s$ .

If during this process a node  $v$  previously added to the routing tree is found, then the process stops and a new phase starts. In this new phase, paths are generated

from  $v$  to the destination  $i$ . The destination  $i$  chooses one of the paths according to a selection function  $F$  (similar to the function used by Kompella et al. [110]), which is defined for each path  $P$  and given by the following expression:

$$F(P) = \frac{C(P)}{\Delta - D_{T_{i-1}}(s, v) - D(P)},$$

where  $C(P)$  and  $D(P)$  are the cost and delay of path  $P$ ,  $\Delta$  is the maximum delay in this group, and  $D_{T_{i-1}}(s, v)$  is the current delay between the source  $s$  and node  $v$ .

### 3.5.2 Minimizing Configuration Time

Every multicast network requires a period during which information is exchanged and the initial connection is performed among the several members of the multicast group. The duration of this period is best known in the literature as the *configuration time*. Although such a period is necessary for the proper operation of the network, it is also desirable that the configuration time be as small as possible.

When a multicast network is allowed to accept dynamic membership, a considerable amount of time is spent on configuration. Moreover, this process can be periodically repeated based on when the system administrators decide that a completely reconfiguration would be useful.

In such situations, it is interesting to find ways to reduce the total time spent on network configuration. With this purpose, Jia [99] proposes a distributed algorithm which addresses the issue of configuration time minimization. A distributed approach is employed there as the basis for the algorithm.

The main strategy used in the algorithm is to integrate routing calculation with network configuration. That is, at the same time that messages are exchanged for the calculation of a routing tree, network configuration is also being performed. In this way, the multicast network can save not only in bandwidth, but also in the total time necessary to setup and calculate the optimal routing tree.

Using this strategy, Jia [99] demonstrated that the number of messages necessary to set up the whole multicast group is decreased. The decrease in the number of messages results in a much faster implementation than would be possible with the employment of distinct phases for configuration and routing tree computation.

## 3.6 An IP Formulation for Multicast Routing

Integer programming may be useful in solving multicast routing problems in several ways. For example, some versions of multicast routing problems have been successfully solved via the use of relaxation and implicit enumeration methods.

An example of this approach to multicast routing was given on Noronha and Tobagi [140], where some types of routing problems were solved using an integer programming formulation. In that paper, a general version of the problem is discussed in which there are costs and delays for each link, and a set  $\{1, \dots, T\}$  of multicast groups. Each group  $i$  has its own source  $s_i$ , a set of  $n_i$  destinations  $d_{i1}, \dots, d_{in_i}$ , a maximum delay  $\Delta_i$ , and a bandwidth request  $r_i$ . In the formulation of the problem, we are also given a matrix  $B^i \in R^{n \times n_i}$ , for each group  $i \in \{1, \dots, T\}$ , of source–destination pairs. The value of  $B_{jk}^i$  is 1 if  $j = s_i$ ,  $-1$  if  $j = d_{ik}$ , and 0 otherwise. The node–edge incidence matrix is represented by  $A \in Z^{n \times m}$ .

The network considered has  $n$  nodes and  $m$  edges. The vectors  $W \in R^m$ ,  $D \in R^m$  and  $C \in R^m$  gives, respectively, the costs, delays, and capacities for each link in the network. The variables in the formulation are  $X^1, \dots, X^T$  (where each  $X^i$  is a matrix  $m \times n_i$ ),  $Y^1, \dots, Y^T$  (where each  $Y^i$  is a vector with  $m$  elements), and  $M \in R^T$ . The variable  $X_{jk}^i = 1$  if and only if link  $j$  is used by group  $i$  to reach destination  $d_{ik}$ . Similarly, variable  $Y_j^i = 1$  if and only if link  $j$  is used by multicast group  $i$ . Also, variable  $M_i$  represents the delay incurred by multicast group  $i$  in the current solution.

In the following formulation, the objectives of minimizing total cost and maximum delay are considered. However, the constant values  $\beta_c$  and  $\beta_d$  represent the relative weight given to the minimization of the cost and to the minimization of the delays, respectively. Using the variables shown, the integer programming formulation is given by:

$$\min \sum_{i=1}^T r_i \beta_c C Y^i + \beta_d M_i \quad (3.1)$$

subject to

$$A X^i = B^i \quad \text{for } i = 1, \dots, T \quad (3.2)$$

$$X_{jk}^i \leq Y_j^i \leq 1 \quad \text{for } i = 1, \dots, T, j = 1, \dots, n, k = 1, \dots, n_i \quad (3.3)$$

$$M_i \geq \sum_{j=1}^k D_j X_{jk}^i \quad \text{for } i = 1, \dots, T, k = 1, \dots, n_i \quad (3.4)$$

$$M_i \leq L_i \quad \text{for } i = 1, \dots, T \quad (3.5)$$

$$\sum_{i=1}^T r_i Y^i \leq C \quad (3.6)$$

$$X_{jk}^i, Y_j^i \in \{0, 1\}, \quad \text{for } 1 \leq i \leq T, 1 \leq j \leq K, 1 \leq k \leq n_i. \quad (3.7)$$

The constraints in the above integer program have the following meaning. Constraint (3.2) is the flow conservation constraint for each of the multicast groups. Constraint (3.3) determines that an edge must be selected when it is used by any multicast tree.

Constraints (3.4) and (3.5) determine the value of the delay, since it must be greater than the sum of all delays in the current multicast group and less than the maximum acceptable delay  $L_i$ . Finally, constraint (3.6) says that each edge  $i$  can carry a value which is at most the capacity  $C_i$ . This is a very general formulation, and cannot be solved exactly in polynomial time because of the integrity constraints (3.7).

This formulation is used in Noronha and Tobagi [140] to derive an exact algorithm for the multicast routing problem. The algorithm proceeds as follows. Initially, a decomposition technique is used to divide the constraint matrix into smaller parts, where each part could be more easily solved. This can be done using standard mathematical programming techniques as shown, for example, in [20]. Then, a branch-and-bound algorithm is proposed to solve the resulting problem. In this branch-and-bound, the lower bounding procedure uses the decomposition found initially to improve the efficiency of the lower bound computation.

### 3.6.1 *Minimizing Bandwidth Utilization*

A frequent problem occurring during the construction of multicast trees is the necessary trade off between bandwidth utilization and total cost of the tree. As we have seen, traditional algorithms for tree minimization try to reduce the total cost of the routing tree. However, this in general does not guarantee an optimal bandwidth utilization. In fact, it is easy to generate trees where some links are heavily used, which increases bandwidth utilization in specific parts of the network.

On the other hand, employing algorithms for bandwidth minimization does not lead to a solution with minimum cost, but the correlation between these two objectives may be closer than in the first case. For example, consider the greedy algorithm described in [74]. The algorithm works by connecting destinations sequentially to the source node. Each destination is then linked to the nearest node already connected to that source. In this way, bandwidth is saved by reusing existing paths.

Fujinoki and Christensen [74] describes an algorithm for maintaining dynamic multicast trees which tries to solve the trade off discussed above. The algorithm, called *shortest best path tree* (SBPT), uses shortest paths as a way to connect sources to destinations. Using this approach, the distance between nodes is defined as the minimum number of edges in the path between them.

The algorithm proceeds as follows. In its first phase, the algorithm consists of computing the shortest path from  $s$  to all destinations  $d_i$ . This provides the necessary information about minimum distances between nodes in the network. As we have previously pointed out, such information can alternatively be collected from unicast protocols, if they are available.

In the second phase, the algorithm performs a sequence of steps for each destination  $d_i \in D$ . Initially, it computes the shortest paths from  $d_i$  to all other nodes in  $G$ . Then, the algorithm takes the node  $u$  which has minimum distance from  $d_i$  and at the same time occurs in one of the shortest paths from  $s$  to  $d_i$ . By making this choice, the system tries to favor nodes that are already in the routing tree and contributing the smallest possible increase in the total cost.

Computational experiments have shown that the procedure above gives solutions that are not only of high quality, but also have small bandwidth utilization when compared to more traditional methods.

### 3.7 The Degree-Constrained Steiner Problem

We have seen several examples of constraints that can be imposed to the multicast routing problem, and how these restrictions can be modeled as a particular form of the general Steiner tree problem on graphs. Each such restriction is necessary to incorporate features of real networks that are frequently left out of theoretical models.

A final type of constraint that we will discuss in this chapter is related to the allowed maximum degree for nodes in the multicast network. It is generally assumed in the general Steiner problem on graphs that the degree of any node in the solution tree is bounded only by the degree of the node in the underlying network. Thus, for example, it could be possible to have a single node connected to all other required nodes, as long as they are connected in the graph. While this is possible in a general network model, it is not how most networks are implemented. In practice, there are physical constraints on the outputs of routers, such as a limit on the number of outbound connections. Such concerns make it necessary to look for models where the number of connections allowed from each single node is limited, instead of accepting solutions that would be infeasible in practice.

To satisfy such degree requirements, it would be interesting to propose a modified version of the multicast routing problem with degree restrictions. When the maximum number of connections from any node in the network is required to be bound by a constant, then we have what is called in the literature the *degree-constrained multicast routing problem*.

For the special case of multicast, the degree constrained version of the multicast routing problem reflects the fact that it may not be possible to make a very large number of copies of the received data at the same time. This is particularly true in the case of high speed switches, where the speed requirements may in practice prohibit an unbounded number of copies of the received information.

For an example of such restrictions in practice, consider ATM networks, where the number of out connections have a fixed upper bound [203]. Solving such a problem within the context of Steiner tree heuristics requires that the exploration of alternative formulations where the degree of nodes in the resulting tree is constrained.

For an example of this kind approach, one can check Bauer [17], where an algorithm for this version of the problem is proposed and analyzed. The algorithm returns a solution that is based on the traditional methods of multicast tree construction. However, the heuristic has a special step that is used to guarantee that the final solution satisfy the given restrictions on the number of adjacent nodes for each router in the network.

The authors of that paper have demonstrated experimentally that such heuristics provide solutions very close to the optimum, for a large number of instances of the multicast routing problem. They have also shown experimentally that, despite the restrictions on node degrees, almost all practical instances have feasible solutions that can be found by means of the proposed heuristics.

### 3.8 Conclusion

In this chapter, we reviewed a number of algorithms for the solution of multicast routing problems. We have targeted our discussion to algorithms and heuristics that use the Steiner tree problem as the starting point to model the multicast network.

As the dominant method to model multicast networks, the Steiner tree has been of great importance for algorithm designers in this area. The existence of efficient heuristics for the Steiner problem have also contributed to the development of both parallel and sequential algorithms for multicast routing.

Despite the advantages of such modeling decisions, it is necessary to remark that the Steiner tree may not be the best way to model multicast routing in some situations. Because of the underlying computational complexity associated with the Steiner tree problem, the proposed solutions have been heuristic in nature.

It might be possible, however, to find optimal solutions for specific classes of the multicast routing problem. In that case, it is possible that other representations might be better suited as a way to design an exact algorithm. We will explore other modeling strategies for multicast routing problems in the following chapters.



## Chapter 4

# Online Multicast Routing

In this chapter, we study a class of multicast routing problems that cannot be described by a static definition of a network and a multicast group. In the online version of the multicast routing problems, we assume that members of a multicast group are able to enter or leave the group as desired.

In that way, online multicast routing requires a constant update of the multicast routing tree as membership in the multicast group changes.

We describe in the next sections a number of approaches for the solution of this problem. In the next section, we provide an introduction to the terminology used in multicast routing. The following sections will discuss possible approaches for the efficient solution of online versions of multicast routing problems.

### 4.1 Introduction

In the previous chapter, we analyzed multicast routing as an off-line problem, in the sense that we consider the demand nodes to be previously determined and fixed. In practice, however, a multicast session is rarely completely determined at the beginning of the transmission. Most applications will require that demand nodes be added or removed during the operation of the multicast network as needed.

For example, a network-based TV system might allow users to enter or leave the network as they desire. Members of an online meeting may also have a similar ability to enter or leave a multicast group as they seem fit.

For such situations, we want to generalize the multicast routing problem to an online model, which can be done in the following way. Given a graph  $G = (V, E)$ , suppose that a multicast group has initial configuration described by  $(s, d_1, \dots, d_k)$ , where  $s$  is the source, and  $d_i$  for  $i \in \{1, \dots, k\}$  are the destinations.

Consider a supplied vector  $R = \{r_1, \dots, r_n\}$  of requests. Each request is a pair  $r_i = (v, \delta)$ , where  $v \in V$  is a node and  $\delta$  is an instruction to either *add* or *remove* destinations.

With these definitions in place, we can see that the online version of multicast routing is a generalization of the problem described in the previous chapter, where addition and removal operations need to be considered. In particular, in the purely online version of the multicast routing problem, the original configuration has  $k = 0$ , that is, no destinations have been assigned, and only the source node  $s$  is known at the start of the algorithm.

## 4.2 Online Multicast and Tree Recalculation

Multicast routing design is typically viewed as a problem on static networks that, while difficult to solve, can be represented using a formulation that considers the whole data as a single instance. The instance is then submitted to an algorithm that is able to find exact or near-optimal solutions for the routing problem.

Online multicast, on the other hand, cannot be solved to optimality by looking only at the configuration of the multicast groups at any single point in time. This means that the concept of optimal routing tree must be adjusted, depending on the type of what characteristics of the network are the most important for the purpose of the application.

One issue that needs to be addressed in the context of online problems is how changes in multicast group membership affect the quality of solutions as new destinations are added or removed. Each new membership required the addition of paths to the network. This process can negatively affect the optimality of a routing tree.

Depending on how an algorithm handles the process of adding or removing members of the multicast tree, the quality of the solution may deteriorate. What is even worse is that the order in which such nodes are added or deleted to the multicast group can greatly affect the final solution.

Given the way new nodes can be added to a multicast group and the consequences of changes in its constitution, one obvious solution is to perform a reoptimization procedure every time there is a change in group membership. On the other hand, recalculation of the complete routing tree at each step is undesirable, due to its high computational complexity.

Thus, solving practical instances of the online version of the multicast routing problem can be seen as a balancing act between a guarantee of optimality provided by frequent runs of a global optimization algorithm and the resulting implementation performance. In the next sections, we provide an overview of possible solutions for this general problem.

## 4.3 Speeding Up Routing Tree Computations

As discussed in the previous section, finding the right trade off between computation time and quality of solution is crucial for the successful development of online algorithms for multicast routing. With this goal, several schemes for speeding up

the recalculation of near optimal routing trees have been devised in the literature. In this section, we discuss a few of most important techniques.

A common approach used on online methods for multicast routing consists of modifying existing algorithms with the goal of avoiding the penalty of recomputing the entire tree at each modification in group membership. For example, a way of achieving this is maintaining a cache that contains the most used paths. This is possible because each modification on the group membership will typically affect only a small number of existing paths in the routing tree. As such, a single modification in group membership does not justify the recomputation of every path leading from source to destination.

A similar scheme was used on [105], where the authors proposed an algorithm to speed up the creation of multicast routing trees. The algorithm can be applied whenever a dynamic change in group membership occurs. Expanding on the simple idea discussed above, the general goal is to maintain caches not only of particular paths, but also of complete multicast trees, that were originally computed for other multicast groups.

With such a cache, the goal is to facilitate the creation of paths between the source and any new node included in the multicast group. With this objective, the algorithm checks some of the previous multicast trees and identifies potential paths that can be used to serve the new destination node. If such a path can be quickly identified, then it is added to the current routing tree.

While such a path generation scheme takes advantage of previous computations, the risk of maintaining an existing solution and using caching schemes is that the quality of the tree may degrade slowly, until it eventually reaches a point when the total cost of the tree is far away from an optimal solution. This happens because each change triggers a local improvement, and global optimality is possibly lost when doing such local modifications. Low quality solutions can eventually emerge in such a situation [155].

### ***4.3.1 Working with High Complexity Algorithms***

Some techniques for creating routing trees have special difficulties when employed on a dynamic membership environment. For example, source tree-based techniques have a hard time with membership changes because with each change a possibly different source node must be used to calculate the new routing tree. Moreover, the new tree must be computed in such a way that it restores service at the requested level.

However, the algorithms necessary to create such a tree are expensive. This makes the resulting approach unsuitable for use with dynamic groups. To overcome such difficulties, algorithms capable of retrieving data from a path cache have been proposed in the literature.

The main purpose of such algorithms is to find similarities between the existing multicast route and the new routes that need to be created with the addition or

removal of members from the multicast group. Based on this cached information, the algorithm constructs a connecting path that uses complete or incomplete paths that have been previously stored in the cache.

Another approach used by researchers and practitioners working with online multicast routing problems is the creation of specialized algorithms. The difficulty of adapting existing source-based techniques to the dynamic scenarios is part of the motivation behind the development of such specialized methods targeting the online version of the problem.

For example, Waxman [193] defines two special purpose heuristics for the online multicast routing problem. The first type of algorithm can be used in situation where a rearrangement of the routing tree is allowed after a number of changes is performed. The second algorithm proposed by Waxman [193] can be used when no such reconfigurations are allowed by the network.

The theoretical model used for both algorithms is based on the online Steiner tree problem, that is, a version of Steiner tree where one needs to construct a tree satisfying the requirements of the Steiner problem, but also subject to the addition and deletion of nodes [97, 176, 194].

### 4.3.2 *Random Graph Models*

One of the possible ways to deal with online problems on graphs is to use a randomized model that takes into consideration the way the nodes are added or removed. When dealing with the online multicast routing problem, a randomized model can yield insights on the structure of the problem that can in turn lead to the development of high performance algorithms.

A *random graph* is a graph where nodes or edges are added according to a probability distribution. In the random edge model, edges are added to a given graph. That is, given a set of nodes  $V[G]$ , we consider a random process in which an edge  $(u, v)$ , for  $u \in V$  and  $v \in V$ , is added according to a probability distribution  $F$ .

In a similar random graph model, nodes are added to a graph and attached to an existing node at random. The resulting graph will have a fixed number of nodes and edges, where the random attachment will result in a graph configuration associated with a probabilistic distribution.

A random graph is generally used to describe a set of probabilistic events on a graph. The rich theory describing the behavior of random graphs was initially explored by Paul Erdős and his collaborators.

Such random models can be used to study the development of routing trees as new members are added or removed from multicast groups. For example, a random graph model for the multicast routing problem was proposed by Waxman [193], where random graphs were used to model the changes in a routing tree due to group membership requests.

Waxman [193] studied how a routing tree must be changed when new nodes are added or removed. To better describe the situation, he proposed a random graph

model where the probability of an edge existing between two nodes depends on the Euclidean distance between them. This probability decreases exponentially with the increase of distance between nodes.

The random inclusion of links to the graph can be used to represent the random process of adding new users to a multicast group. Using this model, Waxman [193] described a greedy heuristic to approximately solve instances generated according to this model.

Another algorithm that can be interpreted in the context of random networks was proposed by Hong et al. [91]. The authors provided a general technique which is capable of handling additions and removals of elements to an existing multicast group. The resulting algorithm uses a model based on the Steiner tree problem and considers some of the common multicast constraints such as maximum delay.

To reduce the computational complexity of each step of the online computation, the authors successfully employed a Lagrangian relaxation technique. According to their experimental results, the algorithm finds solutions that are very close to the optimum when the multicast network is sparse.

### 4.3.3 *Facilitated Inclusion of Group Members*

Another aspect of online multicast routing is to allow members of a multicast group to be added in an efficient way. Efficient insertions may relieve the system from the high complexity associated with tree recomputations. Thus, efficient insertion methods can be beneficial in the long run, since more time is available for improvements in the complete routing tree, if necessary.

Toward this objective, Feng and Yum [65] devised a heuristic with the main goal of allowing easy insertion of new nodes in a multicast group. The heuristic is similar to Prim's algorithm for spanning trees in which at each step it takes a nonconnected destination with minimum cost and tries to add such a destination to the current solution.

The algorithm also uses a priority queue  $Q$  where the already connected elements are stored. The key in this priority queue is the total delay between the elements and the source node. To improve efficiency, the algorithm uses a parameter  $k$  to determine how to compute the path from a destination to the current tree.

Given a value for the parameter  $k$ , the algorithm will compute  $k$  minimum delay paths. Each path starts on the current destination  $d$  and ends on each of the smallest  $k$  elements in the priority queue. Then, the best of the computed paths is chosen to be part of the routing tree.

An interesting feature of the resulting algorithm is that changing the value of the parameter  $k$  will change the amount of effort needed to connect destination. Therefore, just by increasing the value of  $k$  in this algorithm we can obtain better results. The algorithm also facilitates the inclusion of new elements in the multicast network. As a result, the same procedure can be used to grow an existing tree to accommodate new nodes.

## 4.4 Quality of Service Maintenance for Online Multicast Routing

Maintaining quality of service (QoS) during regular transmissions is of great importance on multicast systems. When referring to QoS, the main objective is to provide communication channels where the latency for each user is as small as possible, thus minimizing the transmission delay.

In multicast routing applications, the QoS constraints require that the transmission delay be limited to a certain threshold for each user. The exact delay threshold requested, however, varies by application and by technology employed.

The problem of providing QoS-restricted routes becomes particularly difficult when multiple multicast groups need to compete for access to the underlying network resources. The contention generated by multiple multicast groups can make it difficult to maintain the required QoS. Therefore, the issue becomes one of feasibility for the given set of groups. The problem occurring when multiple multicast groups are present and need to access the same network resources will be investigated in a later chapter.

The literature on QoS for multicast provides some interesting examples of algorithmic approaches for delay minimization. In this section, we will describe in detail the algorithm proposed by Sriram et al. [176]. The algorithm is targeted at the online problem, when multicast members are allowed to dynamically change their membership status on the multicast group.

Thus, we assume that members of such multicast groups may request addition or removal at any time. The goal of the algorithm is to maintain a valid set of routing trees after each of these membership changing events.

### 4.4.1 Problem Definition

To handle the dynamic nature of group membership in the online routing problem, it is necessary to define measures of quality that take into consideration the number of users in each routing tree. An algorithm that includes information about the number of users that enter or leave a multicast group is said to be *adaptive*. Adaptive algorithms are usually better suited to online multicast routing because they add an additional time dimension that can be employed when determining the best routes for a given multicast group.

The controlled rearrangement for constrained dynamic multicast (CRCDM) algorithm [176] creates a shared tree that is dynamically optimized to provide minimum connection cost, while respecting the QoS requirements of delay constraints. The algorithm operates by establishing a measure of quality for the routing tree, called the *quality factor*. As the quality factor is calculated for each group membership change, it can be used to determine if a solution is still acceptable and if a new routing tree must be computed.

The algorithm is able to adapt the routing tree to changes in membership due to inclusions and exclusions of users. To do this, one of the problems the algorithm tries to solve is how to determine the moment in which the tree must be recalculated, and for how long should the algorithm perform modifications to the original tree.

To answer this question, the authors of the algorithm introduced the concept of quality factor, which measures the usefulness of part of the routing tree to the rest of the users. When the quality factor of part of a tree decreases to a specific threshold, that part of the tree must be modified. The authors discuss a technique to rearrange the tree such that the minimum delays continue to be respected.

The algorithm, as proposed in Sriram et al. [176], can be described as follows. Let  $G = (V, E)$  be a network, with associated costs  $c_e$  for  $e \in E$  and delays  $d_e$ , for  $e \in E$ . Each path  $\mathcal{P}$  in the network has costs and delay given by

$$C(\mathcal{P}) = \sum_{e \in \mathcal{P}} c_e,$$

and total delay of

$$D(\mathcal{P}) = \sum_{e \in \mathcal{P}} d_e,$$

where a path is just a sequence of edges  $\mathcal{P} = \{e_1, \dots, e_k\}$ . A path in a tree  $T$  denotes the unique sequence of edges  $\mathcal{P}_T = \{e_1, \dots, e_k\}$  between two nodes  $v$  and  $w$ .

In the CRCDM algorithm, requests are represented as a vector  $R = \{r_1, \dots, r_n\}$ . Each element of the vector consists of a pair  $(v, o)$ , where  $v$  is the node that will be added to the multicast group, and  $o$  denotes the operation to be carried over (either adding or removing a node from the multicast group).

The algorithm proceeds by creating an initial multicast tree that connects all destinations to the source node. Once the initial solution is created, it is updated whenever the solution quality reaches a specific threshold.

Different online algorithms use various thresholds for routing tree recomputation. A previous algorithm, known as ARIES [19], divided the routing tree into regions, and used the number of changes to each of these regions as a threshold that would trigger the recomputation of the routing tree.

The CRCDM algorithm extends this threshold function by considering how each region of the routing tree contributes to the connectivity of destination nodes. This measure is called the *quality factor* of the algorithm, and is defined in such a way that the number of recomputations of the routing tree can be minimized. Without going into further details on the notation used by the algorithm, the quality factor used there can be summarized as

$$Q_R = \frac{N_d(R)}{N(R)},$$

where  $N_d(R)$  is the number of nodes in region  $R$  that are servicing a destination, and  $N(R)$  is the number of nodes in the region.

#### 4.4.2 Updating the Routing Tree

Once a routing tree is created for an instance of the online multicast routing problem, an updating procedure must be used whenever a new node is added to the multicast group. The procedure used by the CRCDM algorithm consists in finding a shortest path between the source and the new destination such that the resulting path satisfies the delay constraints for the destination node.

The strategy used is as follows. Suppose that node  $x$  needs to be added to the routing group. To find the best location where a route to  $x$  can be inserted, the algorithm calculates the delay constrained shortest-path from  $x$  to any other node  $w \in T$ , where  $T$  is the routing tree that was previously calculated. The order in which paths are created is determined by the remaining capacity of each node, that is, nodes with higher remaining capacity are explored first.

After the paths to the new destination node  $x$  are computed, the selection is made based on the following function:

$$SF(P) = \frac{C(P)}{\Delta - D_T(s, v) - D(P)},$$

where  $P$  is a path,  $C(P)$  is its associated cost, and the denominator is the residual delay of the path on the given routing tree. The path with lowest value of  $SF(P)$  is then chosen as the new path that connects  $x$  to the routing tree. After this updating step, the node  $x$  is then considered part of the routing tree  $T$ .

Similarly, the deletion of destinations on the multicast group is done by a simple procedure. First, the algorithm checks that, after the removal of the selected node (say  $x$ ), the remaining path from  $x$  to the routing tree is still part of other routes. It is not, then the set of nodes that are found not to be part of other routing trees is deleted. This is repeated until the resulting tree is minimal and connects the source to all required destinations. After this process is finished, the procedure also updates the quality factor of the routing tree according to the new configuration.

Finally, this algorithm also requires a rebalancing step. This is necessary since, after several insertion and deletion steps, the routing tree can become suboptimal. Notice that the modifications described above to update the routing tree result in a new configuration that is locally optimal with respect to the existing structure. However, it does not guarantee that the continued introduction and removal of new nodes will produce in a near-optimal solution.

To avoid pathological cases where the routing tree is severely suboptimal, the algorithm uses a rebalancing step after a number of iterations. The goal is to run the main optimization procedure when  $k$  iterations of the insertion/removal algorithm has been run. In this way, we can guarantee that the routing tree will go back to an improved state as necessary. The value of  $k$  can be determined empirically, so that we do not spend too much time recomputing the routing tree.

Instead of using a parameter  $k$ , we can instead decide to run the rebalancing step only when the cost of the tree reaches a certain threshold based on the value that was achieved on the original run. Depending on how the network is configured, this may be a more effective way to control changes in the routing tree.

## 4.5 OSPF-Based Algorithms for Online Multicast Routing

A technique that has been frequently employed to speed up the process of creating routes between multicast nodes consists of using the OSPF protocol. In this case, the resulting algorithm uses routing information that is already available in the form of the data propagated by unicast protocols. The use of such information can frequently simplify the resulting multicast routing algorithm.

OSPF-based algorithms are within the context of online algorithms because OSPF data is computer online, as nodes are added to or removed from the network.

For example, a heuristic proposed by Baoxian et al. [15] explores this strategy to provide a solution for the multicast routing problem with delay constraints. The use of OSPF data makes it possible for the resulting algorithm to run within complexity  $\mathcal{O}(|D||V|)$ , where  $D$  is the set of destinations in the multicast group.

The algorithm has two main phases. In the first phase, it checks, for each destination  $d_i$ , if there is a path from the source  $s$  to destination  $d_i$  satisfying the required delay constraint. This step basically determines the feasibility of the instance of the routing problem.

In its second phase, the algorithm uses a heuristic to construct a unicast path from the source  $s$  to each destination node  $d_i \in D$ . This heuristic constructs a path using information provided by the existing unicast protocol. Once the unicast path is retrieved, the list of predecessor nodes is used to construct a full multicast route. The algorithm then combines this information with delay data to construct a feasible solution for the multicast routing problem.

## 4.6 Conclusion

In this chapter, we explored a few online algorithms for multicast routing. Several of these algorithms are based on existing off-line techniques for multicast, which have been adapted for the online setting.

We saw that the online problem is usually stated for sets of multicast groups, instead only one isolated group. Online multicast groups are frequently very dynamic with new additions and deletions requiring a recomputation of the entire tree.

To avoid the cost of tree calculation, the algorithms described in this chapter have introduced heuristics that estimate the amount of necessary change. These algorithms try to avoid total recomputation whenever possible.

Finally, we discussed a class of algorithms that use the underlying information provided by protocols such as OSPF to reduce the amount of computation performed at the multicast level. Some of these algorithms have been successful in blending the speed of unicast routing with the better use of resources provided by multicast routing.



## Chapter 5

# Distributed Algorithms for Multicast Routing

In the previous chapters, we discussed sequential algorithms for multicast routing generation. These algorithms have the ability to process information from a network, including nodes participating in a multicast group. With this information, such algorithms provide a complete solution for the problem, which can be used by network operators to determine the routes to be followed to reach each of the destinations.

It turns out that the multicast routing problem is in fact a distributed problem, since each node considered in the problem has some available processing power. Thus, it is natural to look for distributed algorithms which can use this computational power to reduce the total effort of creating routes.

Many researchers have focused on finding distributed strategies for multicast routing and some of its special cases. In this chapter, we will present some results achieved by distributed algorithms for such problems as computing a delay constrained minimum spanning tree [32, 99].

### 5.1 Distributed Algorithm Concepts

A distributed algorithm is one in which autonomous processing units cooperate to find a solution for a given problem. Given a set of processors  $P$ , each processing unit  $p \in P$  is able to communicate with a subset  $N_P(p)$  of the other processing units.

Sharing of data can be done from starting from a processing node to other nodes in the same network. When the source and destination for the data are not in the same neighborhood, the transfer is performed by traversing a network of nodes. The network is defined by the edges induced by the function  $N_P$ , and assuming that the resulting network is connected.

For simplicity, we assume in this chapter that a network  $N(P)$  formed in this way (e.g.,  $(a, b) \in E[N(P)]$  when  $b \in N_P(a)$ ) is connected. Consequently, there is no pair of processing units that cannot exchange data in a finite number of steps.

When data is directly transferred between two nodes that are neighbors in  $N(P)$ , we say that the data is going through a hop. Using this terminology, we can also say that data can be transferred between any two nodes in the network in a finite number of hops.

## 5.2 Distributed Version of the KMB Heuristic

As a first example of distributed algorithm that can be applied to the multicast routing problem, we discuss a modified version of the KMB heuristic proposed by Chen et al. [32]. In this modified version, the goal is to provide a distributed solution for the Steiner tree problem. If we consider the source and destinations as the required nodes, then we can apply such an algorithm for the multicast routing problem.

The KMB heuristic performs its computation by first using a minimum spanning tree algorithm. The resulting minimum spanning tree is then used to determine routes that have potential for becoming part of the complete solution for the Steiner tree problem.

Finding a minimum spanning tree in a distributed fashion is problem that has been studied by several people. A simple strategy for distributed computation of MST was proposed by Gallager et al. [75].

The algorithm by Gallager et al. [75] works in the following way. It starts with individual nodes, and then extends each group of nodes with another element from the neighborhood. Each separate group is called a *fragment*, and in each iteration a group tries to extend its fragment by adding a neighbor through a link with lowest cost. A set of messages is then defined, so that the numbering of messages can be used to detect if a cycle is being formed.

With the help of the distributed results for MST described above, the algorithm proposed by Chen et al. [32] defines additional messages that can be used to compute a Steiner tree from the MST.

To use such an algorithm to solve a multicast routing problem, we need to create additional checks for constraints appearing on the multicast network, such as delay constraints. Such an algorithm would check if the solution is also feasible for the delay constraints, and in the negative case it would make adjustments to guarantee the requested quality of service (this might involve, for example, finding a distributed shortest path using the delay information).

### 5.3 Distributed Algorithm for Audio and Video on Multicast

Another class of distributed problems on multicast that required attention for its practical importance is the delivery of audio and video. These special cases form the basis for many important applications, including video conferencing.

Kompella et al. [110] proposed some distributed algorithms targeting applications of audio and video delivering over a network, where the restriction on maximum delay plays an important role. In the paper, the objective was to improve the results of a previous centralized algorithm by using a distributed method.

The main objective of the distributed procedure proposed in [110] is to reduce the overall computational complexity associated with the centralized algorithm. It must be noted, however, that by using distributed algorithms some of the global information about the network becomes harder to find (e.g., global connectivity). Thus, it is not always possible to reduce the total complexity of a centralized algorithm through the use of a distributed approach.

In this case, a simplified version of the original nondistributed algorithm has been proposed, in such a way that it does not require use of the connectivity information.

Two versions of the original algorithm for audio and video multicast were provided. The first algorithm in Kompella et al. [110] is basically a version of the well-known Bellman–Ford algorithm [21], which finds a minimum delay tree from the source to each destination. The algorithm has the objective of achieving feasibility, and therefore the results are not necessarily locally optimal. Optimality could be achieved, however, using an intensification phase, such as a local search algorithm.

During the construction of each path, the proposed algorithm verifies the cost of the available edges, and chooses the one with lowest cost that satisfies the delay constraints. The algorithm provides a simple protocol so that each pair of nodes can decide if their common edge will be included in the solution.

Using this method, the algorithm has been reported to return solutions that are within 15–30% of the optimal, for most test instances.

### 5.4 An Adaptation of Prim's Algorithm

In the second algorithm proposed in Kompella et al. [110], the strategy employed is similar to the Prim's algorithm for minimum spanning tree construction. It consists of growing a routing tree, starting from the source node, until all destinations are reached.

The resulting algorithm is specialized according to different techniques for selecting the next edge to be added to the tree. In the first edge selection strategy proposed, edges with smallest cost are selected, such that the delay restrictions are satisfied. The second edge selection rule tries to balance the cost of the edge with

the delay imposed by its use. This is done using a “bias” factor, which gives higher priority to edges with smaller delay, among edges with the same cost.

The bias factor used for edge  $(i, j) \in E(G)$  is

$$b(i, j) = \frac{w(i, j)}{\Delta - (D(s, i) + d(i, j))},$$

where  $D(s, i)$  is the minimum total delay between the source  $s$  and node  $i$ ,  $\Delta$  is the maximum allowed delay, and, as usual,  $w(i, j)$  and  $d(i, j)$  are the cost and delay between nodes  $i$  and  $j$ .

An important issue for distributed algorithms in general is how to determine that the computation has ended. This is a problem because each node operates independently, and therefore there is no centralized way to determine that the goal of the computation was achieved.

The authors of the above mentioned paper also discuss the problem of termination for this multicast routing algorithm. In this case, the problem exists because some configurations can report an infeasible problem, while feasibility can be restored by making some changes to the current solution. A distributed method for restoring feasibility to solution is then proposed as a way of detecting the end of the computation.

## 5.5 Modifications of Dijkstra’s Algorithm

Shaikh and Shin [172] presented a distributed algorithm where the focus is to reduce the complexity of distributed versions of heuristics for the *delay constrained Steiner problem*. In their paper, the authors try to adapt the model of Prim’s and Dijkstra’s algorithms to the harder task of creating a multicast routing tree. In this way, they aim to reduce the complexity associated with the heuristics for Steiner tree, while producing good solutions for the problem.

The methods employed by Dijkstra’s shortest path and Prim’s minimum spanning tree algorithm are interesting because they require only local information about the network, and therefore they are known to perform well in distributed environments. The operation of these algorithms consists of adding at each step a new edge to the existing tree, until some termination condition is satisfied.

In the algorithm proposed by Shaikh and Shin [172], the main addition done to the structure of Dijkstra’s algorithm is a method for distinguishing between destinations and nondestination nodes. This is done by the use of an indicator function  $I_D$  which returns 1 if and only if the argument is not a destination node. The general strategy is presented in Fig. 5.1 (Algorithm).

Note that in this algorithm the accumulated cost of a path is set to zero every time a destination node is reached. This guides the algorithm to find paths that pass through destination nodes with higher probability. This strategy is called destination-driven multicast. The resulting algorithm is simple to implement and, according to the authors, perform well in practice.

|                                 |                                                                      |
|---------------------------------|----------------------------------------------------------------------|
| <b>Dijkstra-based Algorithm</b> |                                                                      |
| 1                               | <b>input:</b> $G(V, E), s$                                           |
| 2                               | <b>for</b> $v \in V$ <b>do</b>                                       |
| 3                               | $d[v] \leftarrow \infty$                                             |
| 4                               | <b>end</b>                                                           |
| 5                               | $d[s] \leftarrow 0$                                                  |
| 6                               | $S \leftarrow \emptyset$                                             |
| 7                               | $Q \leftarrow V$ <b>while</b> $Q \neq \emptyset$ <b>do</b>           |
| 8                               | $v \leftarrow \text{get\_min}(Q)$                                    |
| 9                               | $S \leftarrow S \cup \{v\}$                                          |
| 10                              | <b>for</b> $u \in N(v)$ <b>do</b>                                    |
| 11                              | <b>if</b> $u \notin S$ and $d[u] > d[v]I_D[v] + w(u, v)$ <b>then</b> |
| 12                              | $d[u] \leftarrow d[v]I_D[v] + w(u, v)$                               |
| 13                              | <b>end</b>                                                           |
| 14                              | <b>end</b>                                                           |
| 15                              | <b>end</b>                                                           |

**Fig. 5.1** Modification of Dijkstra's algorithm for multicast routing, proposed by Shaikh and Shin [172]

## 5.6 Using Flooding for Path Computation

Flooding is a simple and useful technique in network routing. Although it is slow in practice, it can be used with success as part of a more sophisticated algorithm. Mokbel et al. [133] discuss such a distributed heuristic algorithm for delay-constrained multicast routing. The algorithm is divided in a number of phases.

The initial phase of the algorithm consists of discovering information about nodes in the network, particularly about delays incurred by packets. In this phase, a packet is sent from a source to all other nodes in the neighborhood. The packet is duplicated at each node, using the flooding technique.

At each node visited, information about the total delay and cost experienced by the packet is collected, added to the packet, and retransmitted. Each destination will receive packets with information about the path traversed during the delay time previously defined. After receiving the packets, each destination then selects the resulting path with lowest cost to be the used path.

As the last step of the initial phase, all destinations send this information to the source node. The source node then will have data about the state of the network that can later be used to determine the best routes to reach each destination.

In the second phase of the algorithm, the source node will receive the selected paths for each destination and construct a routing tree based on this information. Consequently, this is a centralized phase, where existing heuristics can be applied to the construction of the tree.

The complexity of algorithms that use flooding can be punished by the worst case characteristics of this routing technique, unless some restriction is applied to how data is transmitted. In this particular case to improve performance of the algorithm and to avoid an overload of packets in the network during the flooding phase, each

node is required to maintain at most  $K$  packets at any time. Using this parameter, the time complexity of the whole algorithm is  $\mathcal{O}(K^2|V|^2)$ .

## 5.7 Algorithms for Sparse Groups

A common variation of multicast routing occurs when the number of sources and destinations is small compared to number of nodes in the complete network. This is the typical case for big multicast instances, where on average only a few nodes will participate in a group at each moment. In this section we describe an algorithm that explores the sparsity of such multicast trees to help improving results. The algorithm was first described in [175].

The algorithm initially uses information available through a unicast routing protocol to find precomputed paths in the current network. However, problems can appear when these paths induce loops in the corresponding graph. In the algorithm, such intersections are treated and removed dynamically.

The algorithm starts by creating a list of destinations, ordered according to their delay constraints. Nodes with more strict delay constraints have the opportunity of searching first for paths. Each destination  $d_i$  will independently try to find a path from  $d_i$  to the source  $s$ . If during this process a node  $v$  previously added to the routing tree is found, then the process stops and a new phase starts. In this new phase, paths are generated from  $v$  to the destination  $i$ . The destination  $i$  chooses one of the paths according to a selection function  $SF$  (similar to the function used by [110]), which is defined for each path  $P$  and given by

$$SF(P) = \frac{C(P)}{\Delta - D_{T_{i-1}}(s, v) - D(P)},$$

where  $C(P)$  and  $D(P)$  are the cost and delay of path  $P$ ,  $\Delta$  is the maximum delay in this group, and  $D_{T_{i-1}}(s, v)$  is the current delay between the source  $s$  and node  $v$ .

A problem that exists when the multicast group is allowed to have dynamic membership, is that a considerable amount of time is spent in the process of connection configuration. Jia [99] proposes a distributed algorithm which addresses this question. A new distributed algorithm is employed, which integrates the routing calculation with the connection configuration phase. Using this strategy, the number of messages necessary to set up the whole multicast group is decreased.

## 5.8 A Comparison of Distributed Approaches

In this section, we present a comparison of several of the algorithms discussed in the previous sections. We try to summarize the computational complexity of these algorithms in terms of natural parameters for the routing problems they try to solve,

**Table 5.1** Comparison of algorithms for the problem of multicast routing with delay constraints

|   | Algorithm             | Complexity                           | Types of instances                          |
|---|-----------------------|--------------------------------------|---------------------------------------------|
| 1 | Kompella et al. [110] | $\mathcal{O}(n^3)$                   | General online instances                    |
| 2 | Baoxian et al. [15]   | $\mathcal{O}(mn)$                    | Based on unicast information                |
| 3 | Sriram et al. [176]   | $\mathcal{O}(n^3)$                   | Instances with QoS                          |
| 4 | Hong et al. [91]      | $\mathcal{O}(mk(k + T^{\text{SP}}))$ | Dynamic, delay sensitive                    |
| 5 | Kheong et al. [105]   | N/A                                  | General instances, cache must be maintained |

such as the number of nodes in network, the number of links, or the number of members of the multicast group.

The list of algorithms is presented in Table 5.1. In this table, each algorithm is denoted by its published reference, and a description of the type of problem it tries to solve or the major approach employed.

In the algorithm listed in row 4,  $k$  is the number of destinations in the multicast group, and  $T^{\text{SP}}$  is the time required to find a shortest path in the graph. The algorithm in row 5 has no complexity listed because, as explained later, amortized time is the important issue that was not analyzed in the original paper.

The algorithm with lowest computational complexity is the one provided by Baoxian et al. [15]. However, it is important to notice that this complexity is kept low due to the dependence of the algorithm on information provided by unicast routing protocols. Hong et al. [91], however, consider the construction of a complete solution, and have operation on online mode as an additional feature.

Kheong et al. [105] also proposed an algorithm where previous information is reused. In this case, however, the information is provided by previous iterations of the same algorithm. Due to this characteristic, it is difficult to evaluate the complexity of the whole algorithm, since it depends on the amortized complexity on a large number of iterations. This kind of amortized analysis was not carried out in the paper.

Distributed approaches for the Steiner tree problem with delay constraints are more difficult to evaluate in the sense that other features, particular to distributed algorithm, become important. For example, in such algorithms the message complexity, that is, the number of message exchanges, is an important indicator of performance. These factors are sometimes not derived explicitly in the literature.

For the algorithm proposed by Chen et al. [32], the message complexity is shown to be  $\mathcal{O}(m + n(n + \log n))$ , and the time complexity is  $\mathcal{O}(n^2)$ . Also, the worst-case ratio of the solution obtained to the cost of any given minimum cost Steiner tree  $T$  is  $2(1 - 1/l)$ , where  $l$  is the number of leaves in  $T$ .

On the other hand, Shaikh and Shin [172] do not give much information about the complexity of their distributed algorithm. It can be noted, however, that the complexity is similar to that of distributed algorithms for the computation of a minimum spanning tree. Finally, Mokbel et al. [133] derive only the total time complexity of their algorithm, which is  $\mathcal{O}(K^2 n^2)$ , where  $K$  is a constant value introduced to decrease the number of required message exchanges.

## 5.9 Conclusion

In this chapter, we presented an overview of distributed algorithms for multicast routing. Such algorithms are of great practical importance because they use the underlying computational power available in the system to quickly compute solutions for multicast routing problems.

While several approaches have been proposed for the solution of this problem, distributed algorithms still have many facets that have not been explored in the literature. We expect to see many other distributed optimization algorithms that use the inherent distributed power of multicast networks for the solution of the hard optimization problems occurring on routing.

## Chapter 6

# Center-Based Trees and Multicast Packing

Computing routing trees for multicast networks is an expensive operation. As explained in the previous chapters, a number of algorithms for this problem use models borrowed from the Steiner tree problem on graphs, as well as other network problems.

One of the problems with this type of approach is that multicast groups are treated as separate subproblems. Each multicast group is assigned a multicast tree which, even if optimal, may not be the best solution for the problem when all multicast groups are taken into consideration.

Two issues contribute to this problem. First, the routing trees created to serve the given multicast groups are not supposed to share parts of the tree. In practice, it would be very interesting to have at least parts of the routing tree shared among multicast groups to better utilize the available network resources. This issue is considered in the first part of the chapter, when we discuss center-based trees.

The second major problem with the approaches presented up to now is that they ignore the capacity available to multicast groups. If we have knowledge about the existing groups that already use a network, we can adjust the available capacity so that the new routing tree will be valid. A way to solve this problem is using multicast packing models, which will be discussed in the second part of this chapter.

### 6.1 Center-Based Trees

A center-based tree is a routing tree for a multicast routing problem that uses one or more “special” nodes as its root. Instead of solutions routed on the source node, center-based trees can be easier to modify, since the root node may be closer to destinations.

The main advantage of center-based trees, however, is that they can be used by multiple multicast groups. To see this, notice that since the root of the tree is not the source node, it is not totally dependent on the members of a particular group.

Other multicast groups can use parts of the same tree, maybe adding additional paths to connect new members of the group. However, the center node, serving as the root of the tree, will continue as a stable part of the combined routing tree.

Among the proposed schemes for generation of multicast routing trees, center-based trees is one of the most economic because they are able to reuse resources utilized by existing routing trees. The methods for center-based routing, however, differ in how they define the center node. Depending on the definition, one may have a problem that is NP-hard and can be solved only through heuristics. Thus, the resource savings may require a trade off in computation time for this type of multicast trees.

In a center-based trees, network designers also have more opportunity to provide information that can improve and speed up the creation of several multicast groups. The resulting groups can, after that point, coexist and use the network collaboratively.

Routing trees generated as a result of a center-based approach have structural properties that can be explored by algorithms. One of the useful properties that will be discussed in this chapter is the improved flexibility in the exclusion and inclusion of group members.

### 6.1.1 Center-Based Tree Concepts

In this chapter, we present formulations for the problem of generating center-based trees. With this purpose, we initially provide some of the most important definitions used in this area. Then, we discuss how center-based techniques are implemented, along with specific algorithms.

The main objective during the implementation of a center-based strategy is to efficiently combine several multicast routes (serving distinct groups) into a single tree. The use of a single tree provides great flexibility in the management of the members of the multicast network.

The concept of center-based trees was originally developed as a method for improving QoS and minimizing the costs of data transfer on a typical multicast routing tree [188]. The concept has been adopted in several implementations of multicast, due to its practical consequences, specially in dynamic situations, where users need to quickly join or leave an existing routing tree.

Center-based trees have been an important part of the implementation of a few actual multicast protocols. For example, it has been used on the Protocol Independent Multicast (PIM). It was also employed as a central concept in the Core Based Tree (CBT) protocol [166].

When creating a center-based routing tree, the first step is to find a node  $v$  that has a central location in the network. One of the possible ways to define the central node is using the topological center for the set of senders and receivers. The topological center of a graph  $G(V, E)$  is defined as a node  $v \in V$  which is closest to any other

node in the network, that is, a node  $v$  that minimizes  $\max_{u \in V} d(v, u)$ . When  $v$  is calculated, a routing tree rooted at  $v$  may be constructed and used throughout the multicast session.

The basic reasoning motivating the use of this kind of center-based routing tree is that the topological center is a better starting point for the multicast tree. Due to its properties, the location of the topological center is expected to change less frequently than other parts of the tree, and therefore less resources need to be spent on tree computation.

In summary, the scheme used by center-based algorithms departs from the more common idea of rooting the routing tree at the sender. An advantage of this approach is that the resulting routing tree can be easily extended in such a way that can be used by more than one multicast group at the same time.

### 6.1.2 The Topological Center Problem

The topological center problem consists in finding a node  $v$  in a network  $G = (V, E)$  such that the distance from  $v$  to any other node is minimum. That is, we want the node  $v$  that minimizes the function

$$\max_{w \in V \setminus v} d(v, w).$$

The topological center problem is at the heart of the center-based methodology for computing routing trees. However, finding an optimal solution for the topological center problem is NP-hard as demonstrated in Ballardie et al. [14]. Thus, one needs to find reasonable approximations to the topological center problem if one is to use center-based trees as part of a protocol implementation.

A solution proposed by some is to modify the original definition of the topological center problem, so that the resulting problem is not NP-hard. With this objective, one can relax the properties of the topological center, so that it could be more easily computed. Algorithms that use one of these alternative approaches will find a central node that, while not exactly located at the topological center, possess some of its properties. Due to these shared properties, the alternative center node could be used as a relatively good approximation to the topological center.

Algorithms that follow the strategy explained above have been proposed in the literature, including one that uses core points [14] instead of the topological center. Another example of alternative to an expensive topological center computation is the use of rendezvous points, as proposed by Deering et al. [51].

Finding the topological center can be easier in some special cases, for example, if we restrict the problem to find the topological center in a tree. Clearly, there is an exponential number of possible trees that are feasible solutions for the topological center problem. However, for each such tree, it would be easy to find a node satisfying the conditions if we could just test each node in turn. A trivial procedure would then find the center node in  $\mathcal{O}(n^2)$ .

Comparing the mathematical properties of algorithms for the topological center problem is still an issue that requires more research. Through empirical investigation, however, Calvert et al. [29] provided an informative comparison of results using a few center-based tree methods. Notice that in the general case the quality of solutions generated by existing methodologies is not completely understood, other than through computational experimentation.

### 6.1.3 Median and Centroid

As we have seen in the previous sections, there are several ways to define center nodes that will serve as the root for the construction of a center-based tree. Such trees are then classified according to how the central node is chosen from the set of available nodes in the network.

In this section, we present two additional ways of defining a central node in a multicast routing computation. These two ways to define a central node (known as the median node and the centroid) may be used in practical algorithms that explore important properties of center-based multicast tree construction.

First, we define the median node as a node minimizing costs over distances. For formal definition, consider the following. Let the function  $H(v)$  be defined as

$$H(v) = \sum_{u \in V} W(u)d(v, u),$$

where  $W : V \rightarrow R$  is the node weight function and  $d(u, v)$  is the shortest distance between nodes  $u$  and  $v$ .

The *median node* in a tree over the graph  $G = (V, E)$  is defined as the node  $v^*$  that minimizes  $H(v)$ ; that is:

$$v^* = \arg \min_{v \in V} H(v).$$

The median node has some similarities with the topological center, but notice the fact that it represents only an average over the distances occurring in the problem. The minimum is taken over these averaged expression, while in the topological center we want a node that minimizes the maximum distance occurring in the network.

The centroid, on the other hand, is a measure of the weight of all subsets of nodes, when these subsets are generated by the removal of a single node from the tree. A formal definition can be given as follows.

Let  $T$  be a tree over the graph  $G = (V, E)$ . Let  $T_{v,i}$  be the  $i$ th connected subtree generated by the removal of node  $v$  from the original tree  $T$ . Suppose that there are  $k$  such subtrees. The weight of this set of  $k$  subtrees is defined as

$$w(T_{v,i}) = \sum_{u \in T_{v,i}} W(u), \quad \text{for } i \in \{1, \dots, k\}$$

Given this definition, the *centroid* in a tree  $T$  can now be described as the node that minimizes the function  $F$  such that

$$F(v) = \max_{1 \leq i \leq k} w(T_{v,i}).$$

Notice that the centroid is calculated based on the average costs of removing the central node, and as such it is of great importance for the network. The reasoning is that, for this reason, it should be included as the root for the routing trees created by a center-based algorithm.

#### 6.1.4 Properties of Centroids

As shown in the previous section, the centroid is one of the ways to approximate the topological center in a multicast network. Due to the computational complexity of solving the topological center problem, most researchers in this area try to find simplifications and create approximated solutions. By using centroids, core points, rendezvous points, or related methods, it is possible to mostly disregard the NP-complete completeness issue and instead try to find solutions that can be computed efficiently.

In this section, we provide a few properties of the centroid that have been used as a basis for efficient algorithms, as presented in Gupta and Srimani [85].

Using these mathematical properties, one can show that there is an algorithm to compute the centroid that runs in time linear to the number of nodes in the network. First, consider the following theorem (the proofs are not difficult, and are available in [85]).

**Theorem 2.** *Given a network  $G = (V, E)$ , a node  $v \in V$ , and an induced tree of  $G$ , denoted by  $T$ , such that  $v \in T$ . Then the node  $v$  is a centroid if and only if it is also a median of the tree.*

Another property that leads to the creation of efficient algorithms is the following:

**Theorem 3.** *Given a network  $G = (V, E)$ , a node  $v \in V$ , and an induced tree of  $G$ , denoted by  $T$ , such that  $v \in T$ . Then the vertex  $v$  is a centroid if and only if*

$$F(v) \leq 1/2 \sum_{u \in V} W(u)$$

The importance of this theorem is in that any node  $v$  in the network can easily identify if it is a centroid by using the given property. The only requirement is that  $v$  has enough information to determine the sum of node weights for all nodes that are part of the subtree rooted at  $v$ . Once that information is collected in some way (e.g., using an existing unicast protocol), the node can determine if it is a centroid by the use of the previous theorem.

The following two lemmas are easy to prove from the above definitions, and show that, once a centroid  $v$  has been found, other centroids (if they exist) are located in the neighborhood of  $v$ .

**Lemma 1.** *Given a network  $G = (V, E)$  and a vertex  $v \in V$ , then at most one subtree rooted at  $v$  can contain a centroid.*

**Lemma 2.** *Given a network  $G = (V, E)$  and a tree  $T$  induced by  $V$ , then there are at most two centroids in  $T$ . If more than one centroid exist, they are adjacent.*

Finally, we have the following theorem, which can be used to identify if a centroid exist in a given subtree of the original multicast network:

**Theorem 4.** *In a rooted tree where  $x$  is the root, suppose that there are  $k$  subtrees containing  $x$ . Let  $y_1, \dots, y_k$  these root nodes. Then, there is a centroid in the subtree  $y_j$  only if*

$$s_j > \left( \sum_{i=1}^k s_i \right) - s_j,$$

where  $s_i$  is the number of nodes in subtree  $i \in \{1 \dots k\}$ .

The previous theorem can be used in a decomposition-based algorithm. Starting from a root node of the network, decompose a given tree into subtrees containing that node. Then, check if any of these subtrees contain a centroid for the whole network. Then, one can take the subtrees that passed this test phase and continue the procedure, until a centroid is found. The resulting algorithm is highly efficient, as demonstrated by Gupta and Srimani [85].

### 6.1.5 The CBT Protocol

Center-based routing is not only an interesting idea for the calculation of multicast routing trees, but it has also been used to implement multicast protocols that are in use. An example of such a protocol is core-based trees (CBT), which is an established protocol for multicast systems.

CBT was developed with the objective of reducing the costs of maintaining several sources and destinations that operate on the same network. The protocol was first proposed in Ballardie et al. [14], where the authors describe the basic properties of the protocol. The full specification of CBT is freely available as an IETF RFC document [13].

CBT, as any other center-based approach, creates multicast routing trees that are rooted on a center node. The main use of CBT happens when a multicast group has several senders as well as receivers. In that case, CBT is able to use a single multicast routing tree to service all senders in a single multicast group.

In comparison, a traditional multicast routing protocol when receiving requests from multiple sources, would create several routing trees. Each such tree would be

rooted on one of the source nodes pertaining to the given group. CBT, by its mode of operation, is able to avoid additional expense, both in computation time and in network resources, necessary to support as many sources as necessary.

Another advantage of using a single, center-based tree in the CBT protocol is the improved scaling of such an arrangement. Traditional multicast routing approaches may not be able to scale when the network group requires the addition of a large number of sources. In contrast, CBT was designed with scaling as a requirement. Multicast trees generated by CBT can have several source nodes. This is possible since each source will share the same tree, which is rooted on a center node, instead of another source.

## 6.2 The Multicast Packing Problem

Multicast routing is a problem that needs to be considered in the context of other communication activities. As such, it is a problem that should not be studied in isolation, but in the environment where other related issues may be of great importance.

For example, in the general context of multicast applications, several groups of users will be sharing the same connections, and this will have an important impact on the results reached by optimization algorithms when applied to real networks. To achieve good performance on these situations, we need to expand the analysis of the multicast routing problem into the more general setting of multiple groups.

In this section, we study the multicast packing problem, which is primarily concerned with how multiple multicast groups interact and share the underlying network. The resulting problem is modeled using mixed integer programming. We describe the computational techniques used to guarantee that, for such formulations, the constraints that model simultaneous bandwidth use by the different multicast groups are correctly satisfied.

### 6.2.1 Problem Description

A more general view of the multicast routing problem can be found if we consider the required constraints arising when more than one multicast group exists. In this case, there is a number of applications that try to use the network for the purpose of establishing connection and sending information with nodes organized into different groups.

The network capacity must be shared between groups according to their bandwidth requirements. These capacity constraints are modeled in what is called the multicast packing problem in networks. Due to its practical implications, this problem has attracted some attention in the past few years [34, 158, 190].

In the next section, we discuss how this problem can be modeled using an integer programming approach. Such a formulation can be used in several ways, such as finding a relaxation that provides a near optimal solution. Another option is to use a more sophisticated algorithm for integer programming, such as a branch-and-cut strategy, to yield optimal solutions for reasonably sized instances.

## 6.2.2 Integer Programming Formulation

The congestion  $\lambda_e$  on edge  $e$  is given by the sum of all load imposed by the groups using  $e$ . The maximum congestion  $\lambda$  is defined as the maximum of all congestion  $\lambda_e$ , over edges  $e \in E$ . If we assume that there are  $K$  multicast groups, and each group  $k$  generates an amount  $t^k$  of traffic, an integer programming formulation for the multicast packing problem is given by

$$\min \lambda \tag{6.1}$$

subject to

$$\sum_{i=1}^K t^k x_e^k \leq \lambda \quad \text{for all } e \in E \tag{6.2}$$

$$x_e^k \in \{0, 1\}^{|E|} \quad \text{for } i = 1, \dots, K, \tag{6.3}$$

where variable  $x_e^k$  is equal to 1 if and only if the edge  $e$  is being used by multicast group  $k$ .

A number of approaches have been proposed for solving this problem. For example, Wang et al. [190] discuss how to set up multiple groups using routing trees, and formalized this as a packing problem. Two heuristics were then proposed. The first one is based on known heuristics for constructing Steiner trees. The second is based on the cut-set problem. The constraints considered for the Steiner tree problem are, first, the minimum cost under bounded tree depth; and second, the cost minimization under bounded degree for intermediate nodes.

Priwan et al. [158] and Chen et al. [34] proposed similar formulations for the multicast packing problem using integer programming. The last authors considered two ways of modeling the routing of multicast information. In the first method, the information is sent according to a minimum cost tree that runs among nodes in the group, and therefore can be modeled as a Steiner tree problem.

In the second, more interesting version, the information is sent through a ring which visits all elements in the group. This formulation results in a problem similar to the traveling salesman problem. Using these two formulations, the authors describe heuristics that can be applied to compute approximate solutions. Comparisons were made between the two proposed formulations with respect to the quality of their respective solutions to the multicast packing problem.

In the integer formulation of the multicast packing problem, a variable  $x_e$  exists for each edge  $e \in E$ , where  $x_e$  is equal to 1 if and only if the edge  $e$  is selected. Each edge also has an associated cost  $w_e$ . Then, the integer formulation for the tree version of the problem is given by

$$\min \sum_{e \in E} w_e x_e \quad (6.4)$$

subject to

$$\sum_{e \in \delta(S)} x_e \geq 1 \quad \text{for all } S \subset V \text{ such that } m_1 \in S \text{ and } M \not\subset S \quad (6.5)$$

$$x \in \{0, 1\}^{|E|}, \quad (6.6)$$

where  $M$  is the set of nodes participating in a multicast group and  $\delta(S)$  represents the edges leaving the set  $S \subset V$ . The integer program for the ring-based version is given by

$$\min \sum_{e \in E} w_e x_e \quad (6.7)$$

subject to

$$\sum_{e \in \delta(v)} x_e = 2 \quad \text{for all } v \in M \quad (6.8)$$

$$\sum_{e \in \delta(v)} x_e \leq 2 \quad \text{for all } v \in V \setminus M \quad (6.9)$$

$$\sum_{e \in \delta(S)} x_e \geq 2 \quad \text{for all } S \subset V \text{ s.t. } u \in S \text{ and } M \not\subset S \quad (6.10)$$

$$x \in \{0, 1\}^{|E|}. \quad (6.11)$$

Here,  $u$  is any element of  $M$ . The integer solution for this problem defines a ring passing through all nodes participating in group  $M$ . In Chen et al. [34], these two problems are solved using branch-and-cut techniques, after the identification of some valid inequalities.

### 6.3 Multicast Dimensioning

Another interesting problem occurs when we consider the design of a new network, intended to support a specific multicast demand. This is called the multicast network dimensioning problem, and it has been discussed in some recent papers [72, 159, 160].

According to Forsgren and Prytz [72], the problem consists of determining the topology (which edges will be selected) and the corresponding capacity of the edges,

such that a multicast service can be deployed in the resulting network. Much of the work for this problem has used mathematical programming techniques to provide a definition as well as to compute exact and approximate solutions to the problem.

The technique used in [72] has been the Lagrangian relaxation applied to an integer programming model. We assume that there are  $T$  multicast groups. The model uses variables  $x_e^k \in \{0, 1\}$ , for  $k \in \{1, \dots, T\}$ ,  $e \in E$ , which represent if edge  $e$  is used by group  $k$ . There are also variables  $z_e^l \in \{0, 1\}$ , for  $l \in \{1, \dots, L\}$ ,  $e \in E$ , where  $L$  is the highest possible capacity level, which determine if the capacity level of edge  $e$  is equal to  $l$ .

Let  $d_k$ , for  $k \in \{1, \dots, T\}$ , be the bandwidth demanded by group  $k$ ;  $c_e^l$ , for  $l \in \{1, \dots, L\}$ , and  $e \in E$ , be the capacity available for edge  $e$  at the level  $l$ . Also, let  $w_e^l$ , for  $l \in \{1, \dots, L\}$ , and  $e \in E$ , be the cost of using edge  $e$  at the capacity level  $l$ .

Finally, we define  $b \in \mathbb{Z}^n$  as the demand vector, and  $A \in \mathbb{R}^{n \times m}$  as the node-edge incidence matrix. We can now state the multicast network dimensioning problem using the following integer program:

$$\min \sum_{e \in E} \sum_{l=1}^L w_e^l z_e^l \quad (6.12)$$

subject to

$$\sum_{k=1}^T d_k x_e^k \leq \sum_{l=1}^L c_e^l z_e^l \quad \text{for all } e \in E \quad (6.13)$$

$$\sum_{l \in L} z_e^l \leq 1 \quad \text{for all } e \in E \quad (6.14)$$

$$Ax = b \quad (6.15)$$

$$x, z \in \mathbb{Z}. \quad (6.16)$$

In this integer program, constraint (6.13) ensures that the bandwidth used on each edge is at most the available capacity. Constraint (6.14) selects just one capacity level for each edge. Finally, constraint (6.15) enforces the flow conservation in the resulting solution.

The problem proposed above has been solved using a branch-and-cut algorithm, employing standard cut types. The authors also use Lagrangian relaxation to reduce the size of the linear program that needs to be solved. Some primal heuristics have been designed to exploit similarities with the Steiner tree problem. These primal heuristics were used to improve the upper bounds found during the branching phase. As a result, the algorithm has been able to solve instances with more than 100 nodes in reasonable time.

## 6.4 Conclusion

In this chapter, we discussed approaches for multicast routing problems that take into consideration the existence of multiple groups operating in the same network. In this scenario, it becomes less important to find the optimum solution for a specific multicast group. More emphasis is given in this case to the existence of good solutions when multicast groups need to operate at the same time.

The first approach discussed is the center-based approach, where a central node is considered to become the root of the routing tree. Such a routing tree can simultaneously serve multiple multicast groups. The properties used to define the center node make it possible to reuse routes with a minimum of additional computation. While a few possible definitions for the center node, such as the topological center, are NP-hard to solve, there are other ways to find central nodes in polynomial time.

The multicast packing problem is concerned with the determination of multicast routes for simultaneous groups while respecting the capacity constraints of the underlying network. The problem can be established using an integer programming formulation, and standard techniques such as branch-and-cut have been used to find solution for these formulations.

Finally, we discussed the multicast dimensioning problem, where the main issue is finding an optimal design for a network, given the known requirements of the multicast groups. Although such a problem is NP-hard in general, near optimal solutions can be found using standard integer programming techniques, as long as the network is relatively small.



## Chapter 7

# Metaheuristics for Multicast Routing

Approximation algorithms or simple heuristics are the normal way to go when we need fast results for a combinatorial optimization problem. However, when higher accuracy is needed and there is sufficient time to run a more detailed search algorithm, metaheuristics allow us to consistently find better solutions [129]. Such metaheuristics provide a framework for intensification of global search that includes randomization as a basic strategy to explore the set of feasible solutions in an efficient way.

In this chapter, we explore some of the best metaheuristics for combinatorial optimization, especially the ones applicable to multicast and related routing problems. We start with a discussion of some popular metaheuristics, such as tabu search, simulated annealing, genetic algorithms, and GRASP. We give a more detailed example of the use of the GRASP metaheuristic in the area of multicast routing.

### 7.1 Metaheuristics

A metaheuristic is a general framework for the construction of algorithms for optimization problems. Metaheuristics describe not only a single algorithm but also a family of related algorithms which can be used to efficiently provide solutions for a combinatorial problem.

Metaheuristics employ stochastic techniques to achieve its desired results. However, the exact way in which stochastic elements are combined to form a complete algorithm is particular to each metaheuristic.

Two optimization strategies are commonly used in metaheuristics. They are referred to in the literature as intensification and diversification. Intensification is a strategy whereby a solution is continuously improved through the use of small perturbations of the original solution. A well defined perturbation method  $\pi$ , when applied to a single solution  $\sigma$ , generates a neighborhood of  $\sigma$ , according to  $\pi$ , which

we can denote by  $\epsilon(\sigma, \pi)$ . An intensification strategy will apply perturbation  $\pi$  and look for the best solution in the neighborhood  $\epsilon(\sigma, \pi)$  of the current solution  $\sigma$ .

A diversification strategy, on the other hand, tries to generate solutions that are different from the existing ones, using some stochastic method. In its simplest form, a diversification algorithm can produce a complete random solution. A more elaborate procedure, however, would add random elements to an existing solution.

Diversification is the main element that separates metaheuristics from other types of heuristics, such as greedy algorithms. Through diversification, a metaheuristic is able to evaluate a bigger portion of the solution space for the problem, while avoid to stay trapped on nonoptimal solutions.

Another factor of differentiation between metaheuristics is the way solutions are stored in memory. Some metaheuristics maintain a single solution that is randomly changed, while others keep entire sets of solutions, which are sometimes called populations.

Many heuristics try to simulate natural processes, and use them as a guiding method that can be explored to optimize a mathematically defined function. For example, Simulated Annealing tries to emulate the annealing process on metals. As another example, Genetic Algorithms [101] use natural selection as an inspiration for its methods. Tabu Search uses an analogy with gravity, while Ant Colony tries to emulate features of insect communities.

In the following, we describe some of the best known metaheuristics and how they have been used to generate solutions for multicast routing problems.

### 7.1.1 Genetic Algorithms

Genetic Algorithm (GA) is a metaheuristic that tries to simulate the process of natural selection for the solution of combinatorial optimization problems. The basic premise of GAs is that a solution for a problem can be selected among a population of candidates. The populations are cyclically replaced with each new population being called a generation.

A solution is encoded as a sequence, where each element is a problem chromosome. In this context, a chromosome is a binary representation of a component of the problem. For example, if the problem calls for a subset of the arcs in a network, then we can represent this solution as a set of binary variables, each having the value 1 if the corresponding arc is selected. The chromosome in this case is the binary value stored for each arc, and the sequence of chromosome is the complete solution.

At each new generation, solutions are ranked according to their fitness to the objective function. For example, in a combinatorial optimization problem, the fitness can be the same as the value of the objective function. When moving from generation to generation, GA operators are used to select solutions with higher fitness to persist, while low fitness solutions are discarded or recombined into other solutions.

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><b>Genetic Algorithm</b></p> <pre> 1  Initialize population 2  for 1 to number of iterations do 3    for 1 to size of population do 4      Select parents <math>\pi_1, \pi_2</math> with selection operator 5      Apply crossover operator on <math>\pi_1, \pi_2</math>, obtaining <math>\pi'</math> 6      Apply mutation operator on <math>\pi'</math> 7      Insert solution <math>\pi'</math>, and remove a low fitness solution 8    end 9  end 10 Return solution with best fitness.</pre> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**Fig. 7.1** Pseudo-code for genetic algorithm

Candidate solutions are created using stochastic methods, usually being the combination (also called offspring) of other candidate solutions. Each GA implements a number of algorithms called operators, which can transform a single solution or combine pairs of solutions to generate new members of the population. There are several types of operators that are commonly used in GAs, many of them inspired by evolutionary processes:

- *Selection*: The selection algorithm is what provides the bias toward better solutions in the creation of a new population of a Genetic Algorithm. When applied to an element of the population, the selection operator will select a particular element with probability that is proportional to its fitness. Therefore, solutions with higher fitness will have more opportunities to reproduce and generate offspring that have similar characteristics.
- *Crossover*: This operator takes two solutions (parent solution) as input and produces an offspring that is comprised of chromosomes taken from both parents. There are several variations of this operator, but the most common is to take the first half of the solution from the first parent and the second half from the other parent. Depending on the type of problem, some postprocessing needs to be done to achieve a feasible solution satisfying all the problem requirements.
- *Mutation*: The mutation operator has the objective of introducing random changes into the population. Given a small probability as an input parameter, the mutation operator selects a solution with the given probability, and flips a bit in a random place of the solution. For example, in a problem where the objective is to compute a routing tree, flipping a bit will change an arc from selected to nonselected. Similar to the crossover operator, some postprocessing may be required after mutation to guarantee that the resulting solution is feasible.

The GA procedure is summarized in Fig. 7.1. It consists of managing a population of solutions for an optimization problem and using the GA operators to improve the current population, until a number of iterations is complete. The objective is to drive the improvement of the population, until a solution that is close to optimal is found as part of the population. This stochastic process results in a final population, where the member that is the best fit is returned as the solution for the problem.

**Fig. 7.2** Local search algorithm

| Local Search |                                                 |
|--------------|-------------------------------------------------|
| 1            | Let $\sigma$ a solution given as input          |
| 2            | $n \leftarrow 0$                                |
| 3            | <b>while</b> $n < \text{MAXTRY}$ <b>do</b>      |
| 4            | $z = f(\sigma)$                                 |
| 5            | $\sigma' \leftarrow \text{localChange}(\sigma)$ |
| 6            | <b>if</b> $f(\sigma') < z$ <b>then</b>          |
| 7            | $n \leftarrow 0$                                |
| 8            | $\sigma \leftarrow \sigma'$                     |
| 9            | <b>else</b>                                     |
| 10           | $n \leftarrow n + 1$                            |
| 11           | <b>end</b>                                      |
| 12           | <b>end</b>                                      |
| 13           | Return $\sigma$                                 |

Genetic Algorithms has been applied to several optimization problems in telecommunication, and in particular in the area of routing. It is a robust method to finding near optimal solutions, but it provides no guarantees concerning the quality of the final solution. Despite numerous studies on the behavior of GAs, there is no way to show that a solution is optimal for a problem by the use GAs. Therefore, while it is an excellent method for finding high quality solutions, it is not sufficient if one needs to find the true optimal solution for a combinatorial optimization problem.

### 7.1.2 Tabu Search

Tabu search is a metaheuristic that tries to extend the concept of local search to the more general case of NP-hard problems, where local solutions are not necessarily the best solution for a given instance of a problem. Local search is an optimization method that tries to continuously improve a solution using a simple strategy for local improvement, until no further improvement is possible.

A simple description of local search method is presented in Fig. 7.2 (Algorithm). As shown in this algorithm, local search starts with a solution given as input, and repeatedly runs a local change step. If the local change improves the solution, then it is stored as the current solution. If the maximum number of tries is exceeded and no improvement is achieved, then the algorithm stops and returns the current solution as the result.

Local search is also called greed optimization because it tries to always improve a solution. In contrast to local search, tabu search tries to improve its current solution, but if that is not possible it will allow the algorithm to move into suboptimal solutions and record them in a list of tabu moves. The goal is to allow other parts of the solution space to be explored. However, since tabu moves cannot be revisited, this avoids the algorithm to be trapped into a cycle of local minimum, followed by suboptimal solutions, and then returning to the same local optimum.

**Fig. 7.3** Algorithm for simulated annealing

```

Simulated Annealing
1  Let  $\sigma$  a random solution
2  Let  $T$  be the initial temperature
3   $n \leftarrow 0$ 
4   $\tau \leftarrow T$  while  $\tau > 0$  MAXTRY do
5       $z = f(\sigma)$ 
6       $\sigma' \leftarrow \text{localChange}(\sigma)$ 
7      if  $f(\sigma') < z$  then
8           $\sigma \leftarrow \sigma'$ 
9      else
10          $p \leftarrow \exp(f(\sigma) - f(\sigma'))/\tau$ 
11          $r \leftarrow \text{random number between 0 and 1}$ 
12         if  $r \leq p$  then
13              $\sigma \leftarrow \sigma'$ 
14         end
15     end
16     Decrease the value of  $\tau$ 
17 end
18 Return  $\sigma$ 

```

### 7.1.3 Simulated Annealing

Simulated Annealing is another variation of local search that tries to explore randomness during the construction of solutions for optimization problems. Simulated annealing is based on a simulation of physical process of annealing, as present in the cooling of metals.

The process used by Simulated Annealing involves a parameter that simulated the temperature. The current temperature in the annealing process determines the amount of suboptimality that can be accepted in each iteration of the algorithm. At the beginning, the temperature is high, meaning that solutions with objective function value far from the current known solution can be accepted. At each iteration the temperature is reduced, so that the amount of suboptimality is reduced. At the end of the algorithm temperature gets close to 0, and therefore the behavior of the algorithm will be similar to local search in its last phase. A high level description of Simulated Annealing is presented in Fig. 7.3.

### 7.1.4 Greedy Randomized Adaptive Search Procedure

Restarting search heuristics have been very successful in a number of problems. One of the most well-studied restarting algorithms is the greedy randomized adaptive search procedure (GRASP). GRASP is a metaheuristic proposed by Feo and Resende [67] aimed at finding near optimal solutions for combinatorial optimization problems.

| GRASP Algorithm |                                                            |
|-----------------|------------------------------------------------------------|
| 1               | Read instance                                              |
| 2               | Initialize data structures                                 |
| 3               | <b>while</b> termination criterion not satisfied <b>do</b> |
| 4               | $s \leftarrow$ create solution                             |
| 5               | Improve solution $s$                                       |
| 6               | <b>if</b> $s$ is the best solution so far <b>then</b>      |
| 7               | Save $s$                                                   |
| 8               | <b>end</b>                                                 |
| 9               | <b>end</b> Return best solution                            |

**Fig. 7.4** Pseudo-code for GRASP

GRASP is composed of a number of iterations, where a new solution is picked from the feasible set, using a construction algorithm, and subsequently improved, using some local search method. GRASP has been very successful in a number of applications such as QAP [143], frequency assignment [82], and satisfiability. The steps of GRASP are summarized in Fig. 7.4 (Algorithm).

## 7.2 A GRASP for Multicast Routing

The main problem in multicast networks is to determine an optimum route to be followed by packets in a multicast group. This is known as the multicast routing problem (MRP). Several heuristic algorithms have been proposed to solve the MRP, which is of great interest for network engineers. Some of these heuristics were presented in the preceding chapters.

In this chapter, we describe a GRASP for the MRP. First we discuss a greedy algorithm for the construction phase of GRASP. The algorithm is based on the well-known algorithm of Kou et al. [114]. The resulting construction algorithm is used in the implementation of a metaheuristic approach for the MRP. In the GRASP approach described, a restarting procedure is used to apply a greedy algorithm together with a local search method to find near optimal solutions to the MRP.

At the end of this chapter, we present computational results produced by this procedure. Simulation results show that the described technique produces superior results in terms of solution quality, when compared to traditional heuristics. We also discuss some improvements to the GRASP implementation that contribute to reduce the computational time spent by the algorithm and improve the overall quality of solutions.

### 7.2.1 Introduction

Given the complexity of solving exactly the routing problem, a large number of heuristic algorithms have been proposed to find good, nonoptimal solutions

[14, 91, 111, 167, 176, 204]. Such methods, however, lack any guarantee of local optimality. This turns out to be an important shortcoming, specially for instances with a large number of group members, since the efficient use of resources is critical in this case.

The most common way of linking the source to destinations is through a tree spanning the involved nodes (source and destinations). The objectives and additional constraints of the problem may vary depending on the specific application and the respective performance requirements. For example, in real time video applications the quality of service (QoS) constraints require that the delay of transmission be less than some fixed threshold  $\Delta$ . Thus, a basic objective would be to minimize the total cost of the tree, subject to the constraint that all source–destination pairs have delay less than  $\Delta$ . This problem is known as the delay constrained multicast routing problem (DCMRP).

To give a formal description of the DCMRP, let  $G = (V, E)$  be a graph where  $V$  is the set of nodes and  $E$  the set of links between nodes. The source node is represented by  $s$  and the destinations are  $D = \{d_1, \dots, d_k\}$ , such that  $D \subset V$ . There is a cost function  $c : E \rightarrow Z_+$  representing the cost of the links and a delay function  $\tau : E \rightarrow Z_+$ , giving the time elapsed when traversing an edge  $e \in E$ . The problem asks for a set of edges  $E' \subseteq E$  such that  $s$  is connected to every node  $d \in D$  on  $G' = (V, E')$ , the maximum delay is bounded, that is,

$$\max_{d \in D} \sum_{e \in \mathcal{P}_d} \tau(e) \leq \Delta,$$

where  $\mathcal{P}_d$  is the path in  $E'$  from  $s$  to  $d$ , and such that the total cost  $\sum_{e \in E'} c(e)$  of  $E'$  is minimum.

In this chapter, we are interested in solution methods for the delay constrained multicast routing problem. In particular, we propose a new method for computing routing trees for multicast networks using a variation of the KMB heuristic [114]. We also propose to use the resulting heuristic as a constructor in a restarting procedure. The resulting metaheuristic can be viewed as an implementation of GRASP. The strategy is employed to avoid suboptimal solutions produced by existing heuristics.

At the same time, we present techniques applied to the GRASP metaheuristic. We introduce a new method for computing the candidate list, and show that this method is more effective in terms of computational time. The new method, combined with improved search heuristics, can be used to yield fast implementations for the problem considered.

The next sections are organized as follows. In Sect. 7.3, a heuristic for the MRP is proposed. Then, in Sect. 7.3.2, a metaheuristic based on the described algorithm is proposed. Computational results are discussed in Sect. 7.4. Concluding remarks are given in Sect. 7.5.

**KMB Heuristic**

```

1  Input: Graph  $G = (V, E)$ , source  $s$ , and set  $D$  of destinations
2  Construct a complete graph  $K(R, E)$  where the set of nodes is  $R$ .
3  Let the distance  $d(i, j)$ ,  $i, j \in R$  be the shortest path from  $i$  to  $j$  in  $G$ .
4  Find a minimum spanning tree  $T$  of  $K$ .
5  Replace each edge  $(i, j)$  in  $T$  by the complete path from  $i$  to  $j$  in  $G$ .
6  Let the resulting graph be  $T'$ .
7  Compute a minimum spanning tree  $\hat{T}$  of  $T'$ .
8   $r \leftarrow false$ 
9  while  $r \neq true$  do
10    $r \leftarrow false$ 
11   if there is a leaf  $w \in \hat{T}$  which is not in  $R$  then
12     Remove  $w$  from  $\hat{T}$ 
13      $r \leftarrow true$ 
14   end
15 end

```

**Fig. 7.5** KMB heuristic for the Steiner tree problem

### 7.3 Basic Algorithm for the MRP

We start revisiting the algorithm proposed by Kou et al. [114], known in the literature as the KMB heuristic. The main advantage of this algorithm is the fact that it is simple to describe and implement, and yet it gives a performance guarantee of two. In fact, empirical studies show that in most cases the results given by the heuristic are better than this theoretical upper bound. The KMB heuristic is presented in Fig. 3.1 (Algorithm). For convenience, we reproduce it in Fig. 7.5 (Algorithm).

#### 7.3.1 Adapting the Solution to the MRP

The strategy used for solving the MRP consists of modifying the KMB algorithm to account for the additional requirements of the MRP. The formal description of the method is given in Fig. 7.6 (Algorithm).

At the beginning of the algorithm, the initial solution is created by the KMB heuristic. Therefore, the initial result is known to be feasible for the Steiner tree problem, but it can be violating some of the delay or reliability constraints of the problem. The main part of the algorithm tries to satisfy these requirements.

The first part of the algorithm checks the solution and determines if it is feasible for the QoS requirements. If there is any path  $\mathcal{P}$  in the solution such that the total delay is greater than  $\Delta$ , additional steps of the algorithm must be used to “fix” the problem. This is done by running the shortest path algorithm (using, e.g., the Dijkstra’s technique) and substituting it in the solution. Note that the shortest path is computed using delays as costs, to minimize the total delay. We assume that, after running this algorithm, the resulting path  $\mathcal{P}'$  has delay less than  $\Delta$ . If this is not

**Heuristic for MRP**

```

1  Input: Graph  $G = (V, E)$ , source  $s$ , and set  $D$  of destinations
2  Run the KMB algorithm (Algorithm 7.5)
3  Let  $T$  be result of the previous step
4  if there is  $\mathcal{P} \in T$  such that delay  $d(\mathcal{P}) > \Delta$  then
5      Substitute  $\mathcal{P}$  by the shortest path between first and last nodes of  $\mathcal{P}$ 
6      Remove redundant edges not required in the tree
7  end
8  while some edge in  $T$  (on path  $\mathcal{P}_i$ ) uses more  $\beta\%$  of capacity do
9      Create a restricted graph  $G_r$ , where:
10     •  $V(G_r) = V(G)$  and
11     • every edge  $e \in E(G_r)$  has capacity less than  $\frac{\beta}{100}c(e) - c_u(e)$ ,
12     where  $c_u(e)$  is the capacity of  $e \in E(G)$ 
13     Find a shortest path  $\mathcal{P}'$  in  $G_r$ 
14     Substitute  $\mathcal{P}_i$  by the shortest path  $\mathcal{P}'$ 
15     Remove redundant edges not required in the tree
16 end

```

**Fig. 7.6** Heuristic for the MRP

the case, then the instance is clearly not feasible, and the algorithm can terminate. The next step is to substitute the infeasible path  $\mathcal{P}$  by the shortest path found as described. Redundant edges must now be removed whenever needed. This can be done in time  $\mathcal{O}(m)$ , where  $m$  is the number of edges in the graph.

The second part of the heuristic deals with the case when the reliability constrained is violated. If this happens, a similar technique is used to find a new path avoiding the violated edges. However, one needs to be more careful because of the increased possibility of finding infeasible solutions. We propose the creation of a reduced graph, where edges are selected only when they have enough available capacity. This graph is denoted by  $G_r = (V, E')$ , with  $E'$  composed of all  $e \in E$  such that the capacity used by  $e$  is less than  $\beta\%$ . The algorithm now tries to find a new shortest path  $\mathcal{P}'$  linking the extreme nodes of one of the violated paths  $\mathcal{P}_i$  that link  $s$  to a destination. Note that the shortest path  $\mathcal{P}'$  is found over the reduced graph  $G_r$ . Thus, the new path is guaranteed to be feasible for the MRP, due to the way it was constructed. This step is repeated while there is an infeasible edge in the solution. Again, note that we assume that the instance of MRP is feasible, and therefore there is at least one solution satisfying these requirements.

With this construction algorithm in hands, the next logical step is to use it in a more general framework of a metaheuristic. In the next section, a metaheuristic algorithm is proposed, based on the restarting procedure known as GRASP.

### 7.3.2 Metaheuristic Description

The GRASP algorithm is known to be a multi-start method, where at each iteration a new solution is constructed, and subsequently improved. In our implementation, the

|                                 |                                                                    |
|---------------------------------|--------------------------------------------------------------------|
| <b>GRASP Construction Phase</b> |                                                                    |
| 1                               | <b>while</b> solution $s$ is not complete <b>do</b>                |
| 2                               | Order the $k$ existing candidate elements <b>do</b>                |
| 3                               | Select a random $\alpha$ , such that $0 < \alpha \leq k$ <b>do</b> |
| 4                               | Let RCL be the set of $\alpha$ best candidates <b>do</b>           |
| 5                               | Randomly select one of the elements in RCL                         |
| 6                               | <b>end</b>                                                         |

**Fig. 7.7** GRASP construction phase

construction algorithm proposed in the previous section is used for the construction phase. A local search algorithm is employed in the improvement phase.

The construction phase of GRASP is presented in more detail in Fig. 7.7. It traditionally consists of creating the solution step-by-step. At each step, a set of candidate elements is selected, and called the restricted candidate list (RCL). An element of the RCL is selected, and added to the solution. Notice that in this case, the elements are the possible paths in a solution. Although this is the most common method of implementation in GRASP, we see in the next section that this scheme can be improved by the careful use of randomization techniques.

### 7.3.2.1 Improving the Construction Phase

In the traditional implementation of the GRASP construction algorithm, each iteration must construct a list of candidates (RCL), and select one of its elements randomly. The size of the RCL is given by a parameter  $\alpha$ , which frequently is randomly determined.

We propose a new method for the GRASP construction phase that is on average equivalent to the existing method, but which is much more efficient in practice. The method uses the following observation

**Proposition 1.** *Let  $x_1, \dots, x_n$  be an unordered sequence, and  $y_1, \dots, y_n$  the corresponding ordered sequence. Then, to find a random element amongst  $y_1, \dots, y_\alpha$ , for  $0 < \alpha \leq n$ , is on average equivalent to select the best of  $\alpha$  random elements of  $x_1, \dots, x_n$ .*

*Proof.* Given the sequence  $x_1, \dots, x_n$ , and selecting random elements of the sequence, the probability of selecting one the elements in the RCL is  $n\alpha/n = \alpha$ . Consequently, if  $W$  is an indicator random variable defined as 1 whenever the element picked is in the RCL, then  $E(W) = \alpha$ . Therefore, after doing  $k$  independent trials, the average number of elements in the RCL is  $k\alpha$ , by the additivity of expectation. If we have  $k = 1/\alpha$ , then on average there is just one element of the RCL within the picked elements. Now, to know what of these elements is the one in the RCL, we just need to take the smallest one.

The observation above gives a very efficient way of implementing the RCL test, which gives, on average, the same results. Start with the full set  $C$  of candidate

```

GRASP Improved Construction
1  while solution  $s$  is not complete do
2    Select a random  $\alpha$ , such that  $1/n \leq \alpha \leq k$ 
3     $k \leftarrow 1/\alpha$ 
4     $c \leftarrow \infty$  for  $j = 1, \dots, k-1$  do
5      Let  $C_j$  be the list of candidates at this iteration
6      Randomly select one element  $x$  in  $C_j$ 
7       $c_j \leftarrow f(x)$ 
8      if  $c_j < c$  then
9         $c \leftarrow c_j$ 
10        $y \leftarrow x$ 
11     end
12     Insert element  $y$  in the solution  $s$ 
13   end
14 end
15 end

```

**Fig. 7.8** Improved construction for GRASP

elements. Then, at each step generate a value of  $\alpha$ , and pick at random  $k = 1/\alpha$  elements of  $C$ . From the picked elements, store only the one which is the best fit for the greedy function. This method is depicted in Fig. 7.8 (Algorithm).

A clear advantage in terms of computational complexity is achieved by the proposed construction method for GRASP. The best advantage is that, while in the original technique the candidate elements must be sorted, this is not necessary in the proposed algorithm. Moreover, the complexity of traditional construction is dependent on the number of candidate elements. In our method, the complexity is constant for a fixed value of  $\alpha$ . For example, if  $\alpha$  is  $n/2$ , then we need just two iterations to find an element in the RCL, with high probability.

**Theorem 5.** *The complexity of selecting elements from the RCL in the modified construction algorithm is  $n \log n$ .*

*Proof.* A probabilistic analysis of the complexity of the algorithm will be used. The important step that must be accounted for, is the selection of a random element from  $C$ . Initially, let us assume that at each iteration of the for loop, the size of the list is  $n$ . Knowing that at each step the value of  $\alpha$  is chosen from a uniform distribution, we find that the average number  $N$  of elements selected in the for loop is

$$\begin{aligned}
 E(N) &= E\left(\sum_{j=1}^n k\right) = \sum_{j=1}^n E(k|\alpha) \\
 &= \sum_{j=1}^n \int_{1/n}^1 E(k|\alpha = t) dt \\
 &= \sum_{j=1}^n \int_{1/n}^1 1/t dt
 \end{aligned}$$

$$\begin{aligned}
&= \sum_{j=1}^n (\log 1 - \log 1/n) \\
&= \sum_{j=1}^n \log n = n \log n.
\end{aligned}$$

Now, the decrease in the size of the list will not change the complexity of the result, since

$$\sum_{j=1}^n \log(n-j) = n \log n + \log \pi + \log \frac{\csc n\pi}{(n-1)!}.$$

The second term is a constant, and the third goes to zero very fast, as  $n$  tends to infinity because of the value  $(n-1)!$  in the denominator.

### 7.3.2.2 Improvement Phase

GRASP has the advantage of being easy to develop, since it is composed of relatively independent procedures (the constructor and local search phases). It is well suited for applications with existing heuristic algorithms, that can be combined with GRASP to find a better solution.

However, one of the weaknesses of GRASP is its incapacity to integrate good solutions found previously into the current search iteration. Since each iteration will create a completely different solution, there is no information added to the system when a good solution is found.

A method that has been used lately to overcome this problem is called path relinking (PR) [4, 162]. In PR, a subset of the best solutions found is kept in a separate memory, called the elite set. At each iteration, one of the solutions  $s$  will be selected, and a process of comparing the current solution with  $s$  will start. Each component of the solution will be changed to the corresponding value on  $s$ , and after this a local search will be initiated to check for local optimality.

The main idea of PR is that, using an element  $s$  of the elite set and a different starting solution, we can find alternative paths in the solution space leading to improvements in the objective function. At the end of the process, the current solution will have objective function at most equal to the objective function of  $s$ . Due to randomization, however, there is a probability that the algorithm can find a better solution.

We modify the general structure of the GRASP metaheuristic to accommodate Path Relinking. The resulting algorithm is presented in Fig. 7.9 (Algorithm). The main modifications made to GRASP are as follows:

- We have to create and maintain a set of elite solutions. This is shown in lines 8 and 14 of Fig. 7.9 (Algorithm).
- When the elite set is complete, each GRASP iteration should run the Path Relinking routine. This is shown in line 6 of Fig. 7.9 (Algorithm).

```

GRASP with Path Relinking
1   $c^* \leftarrow \infty$ 
2  while stopping criterion not satisfied do
3     $p \leftarrow \text{GreedyRandomized}()$ 
4     $p \leftarrow \text{LocalSearch}(p)$ 
5    if  $|\mathcal{E}| = \text{ELITE\_SIZE}$  then
6      PathRelinking
5    else
8      Insert current solution into  $\mathcal{E}$ 
9    end if  $c(p) < c^*$  then
10      $p^* \leftarrow p$ 
11      $c^* \leftarrow c(p)$ 
12   end UpdateEliteSet
13 end
14 return  $p^*$ 

```

Fig. 7.9 GRASP with path relinking for minimization

```

Procedure UpdateEliteSet
1  Input: current solution  $s$ 
2   $w \leftarrow x \in \mathcal{E}$  such that  $f(x) \geq y, \forall y \in \mathcal{E}$ 
3   $b \leftarrow x \in \mathcal{E}$  such that  $f(x) \leq y, \forall y \in \mathcal{E}$ 
4  if  $f(s) < f(b)$  or  $(f(s) \geq f(w) \text{ and } \min_{x \in \mathcal{E}} \text{diff}(s, x) \geq 3)$  then
5    find  $s_p \| s_p = \arg \min_{x \in \mathcal{E}} \text{diff}(s, x)$ 
6     $\mathcal{E} \leftarrow \mathcal{E} \setminus \{s_p\}$ 
7     $\mathcal{E} \leftarrow \mathcal{E} \cup \{s\}$ 
8  end

```

Fig. 7.10 Procedure UpdateEliteSet

The elite set  $\mathcal{E}$  is maintained in memory as a vector of solutions. We describe now how elements are inserted and removed from  $\mathcal{E}$ . Define the difference  $\text{diff}(s_1, s_2)$  between solutions  $s_1$  and  $s_2$  to be the number of different edges in these two solutions. In the initial phase of GRASP with Path Relinking, the elite set is empty. While we have less than `ELITE_SIZE` elements, the current solution is inserted into  $\mathcal{E}$  if it has a difference of at least 3 to all other elements in  $\mathcal{E}$ . The second phase starts when  $\mathcal{E}$  has `ELITE_SIZE` elements.

In the second phase, new solutions generated by GRASP are inserted into  $\mathcal{E}$  according to the update criterion, which is presented in Fig. 7.10 (Algorithm). If the current solution  $s$  is better than all solutions in  $\mathcal{E}$ , then  $s$  is directly inserted. Otherwise, we require that  $s$  has a difference of at least 3 to all elements in  $\mathcal{E}$ . If this is true, and  $s$  is at least better than the worst solution in  $\mathcal{E}$ , then  $s$  is inserted. To maintain the size of the elite set constant, we remove the solution  $x$  which has the smallest difference  $\text{diff}(x, s)$ , for all  $x \in \mathcal{E}$ .

```

Procedure Path Relinking
1  Input: current solution  $s$ 
2   $s' \leftarrow \text{random}(\mathcal{E})$ 
3  for  $i \leftarrow 1$  to  $m = |E|$  do
4    if  $e_i \in s'$  and  $e_i \notin s$  then
5       $\text{delta} \rightarrow \text{evalij}(s, i, j)$ 
6      if  $\delta > 0$  then
7         $\text{LocalSearch}(s)$ 
8         $\text{SaveSolution}(s)$ 
9      end
10   end
11 end

```

**Fig. 7.11** Procedure Path Relinking

The Path Relinking procedure, shown in Fig. 7.11 (Algorithm), starts with the selection of a random solution  $s' \in \mathcal{E}$ . This solution is called the guiding solution because it is used throughout the procedure to choose the next change to be performed. In each iteration, if an edge  $e \in E(G)$  is in  $s'$  but not in  $s$ , the algorithm will include  $e$  in  $s$  and check whether this brings an improvement. The change in objective function caused by this substitution is found using the function  $\text{evalij}$ . If there is an improvement, then we apply local search in the resulting solution to maintain local optimality. The algorithm does not run local search for nonimproving changes to reduce the total computational effort. If the solution found at the current step is better than the best known solution, then it is saved.

The final result of Path Relinking depends on the objective function value of the solutions found during the algorithm. Let  $s^i$ , for  $0 \leq i \leq n$ , be the solution found at the  $i$ th iteration of the Path Relinking procedure (we note that  $s^0 = s$  and  $s^n = s'$ ). If, given the initial solution  $s$  and the guiding solution  $s'$ , we can find a solution  $s^*$  such that  $f(s^*) < \min\{f(s), f(s')\}$ , and  $f(s^*) \leq f(s^i)$ , for  $1 \leq i < n$ , where  $f(s)$  is the objective value of  $s$ , then  $s^*$  is clearly the best solution and is returned by the algorithm. However, if we cannot find an improvement, then we prefer to return some solution which is different from  $s$  and  $s'$ , but still good in some sense.

We define a local optimum relative to Path Relinking to be a solution  $s^i$ , such that  $s^i < s^{i-1}$  and  $s^i < s^{i+1}$ , for  $1 < i < n$ . If there is no such solution, then we define  $s^0$  and  $s^n$  as the only local optima. We define then the resulting solution as  $s^* = \min\{s^i : s^i \text{ is a local optimum relative to Path Relinking}\}$ . In this definition, if there is an improvement, then  $s^*$  is the best improvement found. Otherwise, if there is some local optimum, we choose the best local optimum. Note that we return the minimum of  $s, s'$  only if no other local optimum was found during the execution of Path Relinking. This is done to improve diversity of GRASP, avoiding the unnecessary repetition of solutions already in the elite set.

The complexity of the Path Relinking procedure is determined by the operations performed to find the best improvement. The  $\text{evalij}$  procedure (line 7 of Fig. 7.11 (Algorithm)) has complexity  $\mathcal{O}(n)$  because it simply verifies the impact of the

```

Final Path Relinking
1  while best solution can be improved do
2    for  $i = 1$  to  $n$  do
3      for  $j \leftarrow 1$  to  $n$  do
4        if  $i \neq j$  then
5           $e_1 \leftarrow$  element  $i$  of  $\mathcal{E}$ 
6           $e_2 \leftarrow$  element  $j$  of  $\mathcal{E}$ 
7          Path-Relinking( $e_1, e_2$ )
8        end
9      end
10    end
11  end

```

**Fig. 7.12** Final Path Relinking

change in the solution. Thus, the step with biggest complexity is the local search procedure (line 13). A faster local search implementation was employed with  $\mathcal{O}(n^2)$  complexity. This implies a complexity of  $\mathcal{O}(n^2)$  inside the **for** loop (lines 4–17). Therefore, we have the following result.

**Theorem 6.** *The worst case performance of Path Relinking as described in Fig. 7.11 (Algorithm) is  $\mathcal{O}(n^3)$ .*

### 7.3.2.3 Reverse Path Relinking and Post-processing

The Path Relinking procedure described above can be further generalized, by considering that it can be run not only from the current solution  $s$  to a solution  $s'$  in the elite set, but also in the reverse direction. The results obtained with this change are not necessarily equal to the previous results. We call this modification of the path relinking procedure reverse path relinking. Therefore, in our implementation, for each GRASP iteration the reverse path relinking is also applied, to improve the results of the intensification phase.

As a last step of GRASP, we use Path Relinking as a post optimization procedure at the end of the program. We call this final path relinking. The post optimization step ensures that the solution returned by GRASP is optimal with respect to the path relinking operation. A description for this step is given in Fig. 7.12 (Algorithm). At each iteration of the **repeat** loop in line 1, we run Path Relinking for each pair of solutions  $s_i, s_j \in \mathcal{E}$ . The algorithm stops when, after doing this for all possible pairs in  $\mathcal{E}$ , the objective function of the best solution cannot be improved.

### 7.3.2.4 Efficient Implementation of Path Relinking

One of the computational burdens associated with the Path Relinking method is the requirement of performing some kind of local search for new solutions found during

its execution. This is done to maintain local optimality and further explore the new path in the search space. However, the use of local search can make each iteration of GRASP slower and have a negative effect in the overall computational time.

To avoid this situation, we improved the original local search used in GRASP by using a nonexhaustive improvement phase. In our implementation, only one of the edges not included in  $s$  (selected randomly) is verified and included. This reduces the complexity of local search by a factor of  $n$ , leading to a  $\mathcal{O}(n^2)$  implementation. This scheme is used in each iteration inside Path Relinking.

To enhance the quality of local search outside Path Relinking, we use the following additional method. The modified local search is executed as discussed above. Then, we change randomly two pair of elements in the solution and return to local search as before. The result of this operation allows the algorithm to explore a different, but closely related neighborhood, which can bring further improvements to the current solution. We continue with the local method, until a terminating criterion is satisfied (we used number of iterations without improvement as the terminating condition).

These changes represented a good improvement over the original local search, both in terms of time as well as quality of solutions found, as discussed in the next section.

## 7.4 Experimental Results

The heuristics proposed in this chapter have been implemented using the ANSI-C language. The implementation was devised to be highly portable across platforms and easy to reuse. The compiler employed was the GNU GCC optimized compiler. The level of optimization chosen was `-O2`. The processor used was a Pentium4 with 2.8 GHz, and 512 MB of available memory.

Table 7.1 presents a summary of the results found by the algorithm proposed in this chapter. The first three columns describe the instances used for testing. We employed random instances of the MRP ranging from 100 to 200 nodes and 500–2,000 edges. For each size of instance, we made 50 runs with different instances of the same size. The objective of this methodology is to find results that represent the average behavior of instances.

The results presented in Table 7.1 show that the algorithm implemented as proposed above gives a large improvement over the simple KMB heuristic. We see that for most problems, the solution returned by our heuristic is about 30% better than the KMB results. This gives a clear indication that our results are much closer to the global optimum. In column 9, we decided to report just the time spent in the construction phase, since this is the only comparable part of the two procedures, in terms of computational time. The metaheuristic was, in fact, run for a maximum of 10 min, for each iteration. The same results appear also in Fig. 7.13.

**Table 7.1** Results of running the KMB heuristic and the GRASP metaheuristic on selected instances of the multicast routing problem

| Instance |      |    | KMB       |            |           | Metaheuristic |            |           |       |
|----------|------|----|-----------|------------|-----------|---------------|------------|-----------|-------|
| n        | m    | D  | Best Sol. | Avg. Value | Avg. Time | Best Sol.     | Avg. Value | Avg. Time | Dist. |
| 100      | 500  | 10 | 77        | 93.2       | 1.5       | 59            | 63.3       | 1.8       | 32%   |
| 100      | 600  | 10 | 56        | 69.4       | 1.5       | 42            | 43.6       | 1.9       | 37%   |
| 100      | 700  | 10 | 66        | 74.2       | 1.2       | 43            | 46.8       | 1.3       | 37%   |
| 120      | 800  | 12 | 80        | 87.4       | 2.4       | 50            | 54.1       | 2.9       | 38%   |
| 120      | 900  | 12 | 73        | 91.3       | 2.1       | 60            | 64.3       | 2.5       | 30%   |
| 120      | 1000 | 12 | 56        | 63.6       | 2.2       | 41            | 43.1       | 3.5       | 32%   |
| 140      | 1000 | 14 | 82        | 98.2       | 3.1       | 61            | 65.4       | 3.6       | 33%   |
| 140      | 1100 | 14 | 65        | 81.1       | 3.8       | 56            | 57.5       | 3.7       | 29%   |
| 140      | 1200 | 14 | 65        | 74.9       | 3.3       | 47            | 50.6       | 3.9       | 32%   |
| 160      | 1300 | 16 | 108       | 121.6      | 4.8       | 78            | 80.7       | 4.9       | 34%   |
| 160      | 1400 | 16 | 97        | 112.3      | 4.8       | 70            | 77.7       | 5.6       | 31%   |
| 160      | 1500 | 16 | 89        | 97.1       | 4.9       | 55            | 64.2       | 5.8       | 34%   |
| 180      | 1600 | 18 | 101       | 115.3      | 6.4       | 73            | 80.3       | 6.9       | 30%   |
| 180      | 1700 | 18 | 113       | 122        | 6.5       | 75            | 79.4       | 7.2       | 35%   |
| 180      | 1800 | 18 | 99        | 110        | 6.5       | 69            | 73.4       | 8.4       | 33%   |
| 200      | 1800 | 20 | 119       | 133.6      | 8.7       | 87            | 91.4       | 9.3       | 32%   |
| 200      | 1900 | 20 | 114       | 126        | 8.7       | 79            | 84.6       | 9.7       | 33%   |
| 200      | 2000 | 20 | 112       | 122.5      | 8.6       | 77            | 83.0       | 10.2      | 32%   |

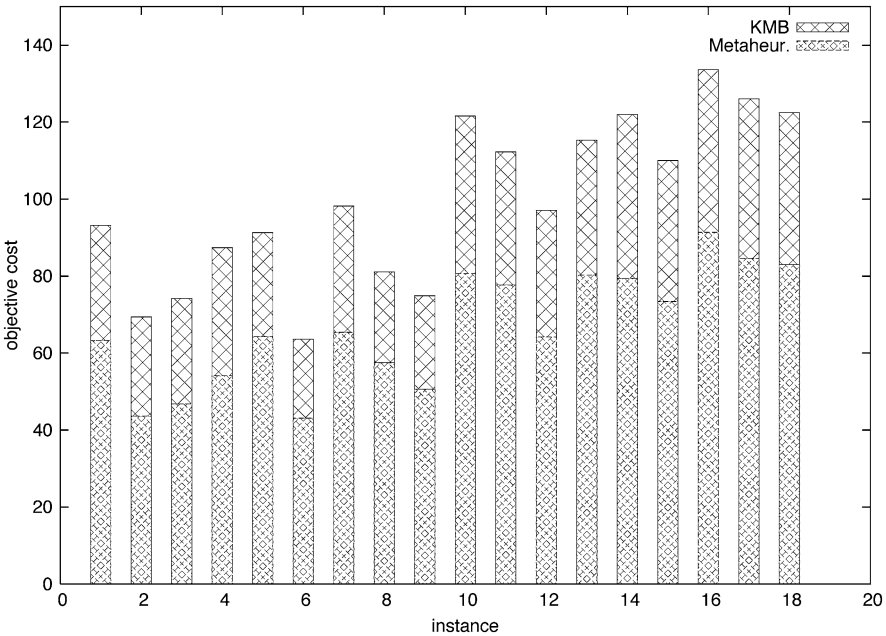


**Fig. 7.13** Comparison between the average solution costs found by the KMB heuristic and our algorithm

## 7.5 Conclusion

In this chapter, we presented a new heuristic algorithm to solve the MRP. Its main technique is to use a modified construction method, based on the KMB heuristic [114] to find feasible solutions for the MRP. We combine this construction algorithm with a restarting procedure based on GRASP.

The resulting algorithm was found to yield near optimal solutions, with very good improvements, compared to the original KMB heuristic. A topic for further research would be to compare this solution to other existing methods. Another interesting topic would be to study distributed versions of this procedure. This would not be very difficult, since it is well known that restarting procedures such as GRASP are simple to parallelize. However, this is important task, since distributed algorithms are essential for the practical implementation of network routing algorithms.

# Chapter 8

## The Point-to-Point Connection Problem

### 8.1 Introduction

We turn our attention to a model that can be used to represent groups of multicast users. The main assumption in this model, known as the point-to-point connection problem, is that there are as many senders as receivers of data in the multicast group.

The point-to-point connection (PPC) problem is an NP-hard, combinatorial optimization problem with applications on several areas. It can be viewed as a natural extension of the Steiner tree problem for networks. Among other applications, the PPC problem can be used to model multicast scenarios in which a common routing tree is used to connect the same number of source and destination nodes.

As an example of a scenario where a PPC model might be useful, consider video conferencing. In such applications, each node in the multicast networks can both send and receive data. Such applications can benefit from the use of a shared multicast tree model supporting data that originates from multiple points.

This chapter is organized as follows. In Sect. 8.2, we introduce the problem. In Sect. 8.3, we provide a more detailed description of the point-to-point connection problem, as well as its use in applications. We also briefly discuss some of the complexity results for the PPC problem and its variants. In Sect. 8.4, the Asynchronous Teams (A-Teams) methodology is discussed in detail. We present a polynomial complexity heuristic algorithm to the PPC problem in Sect. 8.5. In Sect. 8.6, a parallel implementation of the A-Teams is discussed. In Sect. 8.7, computational results found using the algorithm are presented and analyzed. Finally, in Sect. 8.8, we present some concluding remarks.

### 8.2 Point-to-Point Connection and Multicast

The common definition of the multicast routing problem states it as finding routes from one source to several destinations. This is also commonly referred to as point-multipoint routing. On the other hand, situations arise when there may be multiple

sources of available data. It is interesting to pose the question of how to minimize data transmission costs when the problem has this special structure. In this chapter, we study a special version of this general problem, called the PPC problem.

More formally, *multicast routing* is defined in this chapter as the process of sending information from one or more sources to multiple receivers in a network, with the use of a simple “transmit” operation (see [117]). In such situations, three types of nodes can be distinguished, namely *source nodes*, from which data originates; *destination nodes*, which receive data; and *link nodes*.

All source, destination, and link nodes can be used to route information from sources to destinations. Thus, depending on the structure of the network, there is a large number of possibilities on how to route data. Consequently, optimization problems may be formulated based on the many possibilities and on particular optimization goals.

In the PPC, we are given a graph  $G = (V, E)$  and two sets: the set  $S$  of sources and the set  $D$  of destinations, each with the same number of elements. A feasible solution for the problem is one in which each source is connected to at least one destination. Each arc in the network has an associated cost, and the cost of a solution is given by the sum of all costs for the arcs used to connect sources to destinations. The objective of the problem is finding a feasible solution with minimum cost.

Similar to the Steiner tree problem in the PPC, we are interested in a subset of edges forming a forest, that is, no cycle is accepted in the final solution. The sources and destinations play the role of the set of required nodes in the Steiner tree problem. Other nodes in the network can act as transshipment nodes for traffic origination at the sources, which are represented in the Steiner problem as Steiner nodes.

In the remaining of this chapter, we concentrate our attention on algorithms for the PPC problem. After formalizing the problem in the next section, we present a parallel asynchronous methodology to solve it, called A-Teams. We also demonstrate an implementation of this methodology and present computational results obtained by the algorithm.

### 8.3 Problem Formulation

In this section, we formally define the PPC problem. The PPC problem occurs not only on multicast routing, but also in other areas, such as VLSI and vehicle routing [121]. From the optimization point of view, we need to characterize the solutions and the objective function.

Let  $G(V, E)$  be a graph where  $V$  represents the set of devices connected in a network and  $E$  represents the physical links (cables, satellite connections) between these devices. We assign a quantity  $w(i, j)$  to each link  $(i, j)$  in the network. This quantity will represent real costs of using the link. It can also model the time required for delivering a message through the link if that is an important factor.

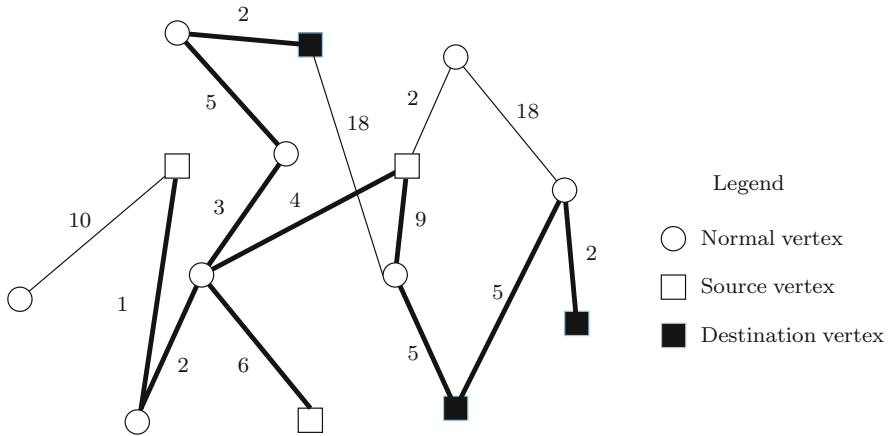


**Fig. 8.1** Instance of the PPC. The *thick lines* represent the best solution, with value 44

We assume that costs of network links are fixed (a situation that is not uncommon in static routing). In addition, we assume that every link has the necessary throughput to convey the desired data.

Moreover, we assume that the number of source and destination nodes is identical. Situations in which the number of sources is less than that of receivers can be easily handled if we include dummy source nodes into the network.

A solution is characterized by a forest  $E' \subset E$ , composed of a set of paths joining the source to the destination nodes (see Fig. 8.1). The cost of a solution is the sum of the costs of all links involved in this solution. The goal of the problem is to find a tree of minimal cost linking the source and destination nodes.

In mathematical terms, let  $|V| = n$  and  $|E| = m$ . For a given positive integer  $p$ , define  $S \subset V$  and  $D \subset V$  to be the set of source and destination nodes, respectively, where  $|S| = |D| = p$  and  $S \cap D = \emptyset$ . A *source–destination coupling* (or *coupling*, for short) is a one-to-one mapping from  $S$  to  $D$ . Then a *PPC*, also referred to simply as *connection*, is a subset  $E'$  of  $E$  such that there is a coupling in which each source–destination pair is connected by a path containing only edges in  $E'$ . Notice that these paths may intersect. Denote the cost of  $E'$  by

$$f(E') = \sum_{(u,v) \in E'} w(u,v).$$

The objective in the undirected and nonfixed *point-to-point connection problem* (PPCP) is to find a minimum cost PPC.

We can devise four different variants of the PPC problem. Besides the undirected, nonfixed version defined above, we can make the graph  $G$  directed or undirected. We can also fix destinations, that is, define a mapping  $\mu$  such that for each source node  $s_i$  there is a fixed destination  $\mu(s_i)$ . In the latter case, the fixed coupling defined by  $\mu$  is given as input.

Li et al. [121] proved that all four versions of the problem are NP-hard. In the same work, the authors proposed a dynamic programming algorithm with time complexity  $O(n^5)$  for the fixed destinations case of the problem on directed graphs when  $p = 2$ . More recently, Natsu and Fang [138] proposed another dynamic programming algorithm, still for the case  $p = 2$ , with overall time complexity  $O(mn + n^2 \log n)$ . They also presented an algorithm for the fixed destinations case when the graph is directed and  $p = 3$ . This algorithm has time complexity  $O(n^{11})$ .

Goemans and Williamson [79] presented an approximation algorithm for the PPC problem that runs in  $O(n^2 \log n)$  and gives its results within a factor of  $2 - 1/p$  of an optimal solution. In the case in which  $p = 1$ , it is easy to see that the problem is equivalent to the shortest path problem.

### 8.3.1 Mathematical Programming Formulation

To derive a mathematical programming formulation of the PPC problem, let us define the following variables:

$$x_{i,j} = \begin{cases} 1 & \text{if } (i, j) \in E' \\ 0 & \text{otherwise} \end{cases}$$

Using this notation, the PPC problem can be stated as:

$$\min \sum_{i=1}^n \sum_{j=1}^n x_{i,j} w_{i,j}$$

such that

$$\sum_{i \in A} \sum_{j \notin A} x_{i,j} \geq 1 \quad \forall A \subset V(G), \text{ whenever } |A \cap S| \neq |A \cap D|$$

$$x_{i,j} \in \{0, 1\} \text{ for all } i, j = 1, \dots, n,$$

where  $S, D$  are, respectively, the source and destination sets discussed above.

As discussed above, although there exist polynomial time dynamic programming algorithms for solving PPC instances, they only apply when the number of source–destination pairs is not greater than 3. In real-world applications, we need to solve far larger problem instances, with many source–destination pairs.

In this chapter, we will describe a stochastic, near-optimal procedure to solve instances of the undirected, nonfixed PPC problem in an acceptable amount of computation time. To do this, we describe the design of teams of heuristics that cooperate to produce good solutions. These heuristic organizations are known as an A-Team, as show in the next section.

## 8.4 Asynchronous Team Algorithms

An A-Team is an organization of heuristics that communicate with each other by the means of exchanging solutions through a shared memory. Each heuristic can make its own choices about inputs, scheduling, and resource allocation. This method was proposed by Simon Doar [173] and has been applied successfully to numerous problems, such as the Traveling Salesman [173], Set Covering Problem [122], Job Shop Scheduling [30, 33], and Constraint Satisfaction [83].

The simple ideas comprising the implementation of an A-Team make it suitable to solve a large number of combinatorial optimization problems. The characteristics of this metaheuristic become even more interesting when there are only a few, simple known heuristics for a problem. In such a case, the existing heuristics can be easily combined into an A-Team to provide improved results. Even for problems for which there are good known heuristics, this method can be used to combine the strengths of different strategies.

The remaining of this section is devoted to the definition of A-Team algorithms. The next section contains details about its implementation for solving the PPC problem.

### 8.4.1 Formal Definition

An Asynchronous Team can be defined as a set  $M = \{M_1, M_2, \dots, M_l\}$  of shared memories and a set  $H = \{H_1, H_2, \dots, H_k\}$  of heuristics. Each memory  $M_i$  holds a set of potential solutions  $S_i = \{s_1, s_2, \dots, s_{k_i}\}$ . Each heuristic  $H_j$  has access (reading, writing, or read-writing) to a subset  $M_l, l \in R(j)$  of the memories, where  $R(j) \subset \{1, \dots, k\}$  is the *reach* of heuristic  $H_j$ . The heuristics  $H$  work as asynchronous strategies, capable of reading and writing new solutions to and from its associated memories. The general metaheuristic design is that the memory holds a population of solutions, which is continuously modified by heuristics. Each heuristic can be very simple, but must contribute different features to the pool of solutions. The content of memories can, therefore, evolve in the sense that the objective function of the best solution is improving.

Among the set  $H$  of heuristics, we can identify some with specific attributes. For example, *construction heuristics* work (alone or in cooperation with other heuristics) to create a new solution. They are used mainly to fill the memories at the beginning of the process.

*Improvement heuristics* are used to modify existing solutions. An improvement heuristic reads a solution from any of the memories to which it has access, executes an improvement procedure and then writes the resulting solution again to one of the accessible memories. *Destruction heuristics* act every time it is necessary to discard a solution from memory. This happens whenever a new solution must be written in an already filled memory.

---

**Algorithm 1** Sequential algorithm for an A-Team
 

---

**Sequential A-Team**

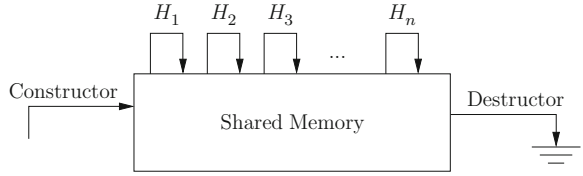
```

1: for each of the memories  $M_i \in M$  do
2:   for  $j \leftarrow 1$  to  $k_i$  do
3:     Call one of the constructors to create solution  $s_j$ 
4:     write( $s_j, M_i$ )
5:   end for
6: end for
7: while termination_condition  $\neq$  True do
8:    $H_i \leftarrow \text{random}(H)$ 
9:    $l \leftarrow \text{random}(R(i))$ 
10:   $s_j \leftarrow \text{read\_solution}(M_l)$ 
11:   $s_k \leftarrow H_i(s_j)$ 
12:  {Call a destructor to remove a solution from  $M_l$ }
13:  destructor( $M_l$ )
14:  {Write the resulting solution}
15:  write( $s_k, M_l$ )
16:  {Update best solution}
17:  if  $f(s_k) < f(\text{best-solution})$  then
18:    best-solution  $\leftarrow s_k$ 
19:  end if
20: end while

```

---

**Fig. 8.2** Example of an A-Team. *Arrows* represent heuristics and the *rectangle* represents shared memory



As the heuristics described can run concurrently, the execution flow of an A-Team is inherently parallel. An equivalent sequential version of the procedure is presented in Algorithm 1.

In Fig. 8.2, a simple A-Team is presented, where arrows represent heuristics and the rectangle represents the only shared memory  $M = M_1$ . One of the heuristics is marked as a constructor, which is represented differently from other heuristics.

The destructor also has its own representation. In this case, the memory  $M_1$  is shared by all heuristics in  $H$ . In any A-Team, the operations that the heuristics are allowed to perform on  $M$  are two, namely *reading* and *writing*. The reading operation does not take any input, and returns a solution stored in  $M$ , selected using some arbitrary criterion. The input of the writing operation is a solution, which is possibly written in  $M$  according to another arbitrary criterion. If a solution  $s$  is to be written, then another solution in  $M$  is selected to be deleted and replaced by the contents of  $s$ .

### 8.4.2 Characterization of an A-Team

We characterize A-Team algorithms predominantly by the following three features. First, the heuristics are *autonomous*, which means that each of them is able to run independently from the others. Second, the shared memory runs *asynchronous* communications, which allows concurrent reading and writing operations from distinct heuristics without synchronization among them (other than any synchronization needed to avoid memory corruption). Finally, heuristics retrieve, modify, and store information continuously in the shared memory until a certain termination condition is achieved.

An execution of an A-Team can be described by a set of events. Each *event* is a 5-tuple composed by the following elements:

1. The time at which the event occurs
2. The input data, if any
3. The state of the shared memory prior to the occurrence of the event
4. The state of the shared memory after the occurrence of the event
5. The output data, if any

According to this formalism, there is a restricted number of types of events occurring during an A-Team execution:

- *The initialization of global variables and the shared memory.* This type of event occurs just once at the beginning, and takes no input. The state prior to the event is undefined, the state after the event contains the initial state of the global variables and the shared memory. No output is generated.
- *The reading operation.* Again, this type of event takes no input. The state does not change, and the output is a solution in  $M$ .
- *The execution of a heuristic  $H_j$ .* It takes one solution in  $M_j$ ,  $j \in R(i)$  as input and possibly changes the state of the global variables. The output of this type of event is also a solution.
- *The writing operation.* A solution is taken as input to be written in  $M$ . This may change the state of the solutions memory, but no output is generated.

Finally, the termination of the A-Team is decided by the algorithm `termination-condition`, as shown in Algorithm 1.

It is clear from the previous observations that there exist several distinct executions for a given A-Team. The solution obtained in each execution may differ. An execution is said to be *acceptable* if it leads to a near-optimal solution. Again, an acceptable execution of an A-Team is not unique. Thus, an acceptable execution is *efficient* if it is performed fast. Finally, an A-Team algorithm is considered *efficient* if it has efficient executions and these occur with high probability.

In the next sections, we show in detail a parallel A-Team algorithm for the PPC problem, and we check its efficiency experimentally.

## 8.5 Solving the PPC Problem

In this section, we describe the variables and heuristics of our A-Team for the PPC problem. The heuristics are divided in classes. We use *construction* heuristics ( $C^*$ ) that build a feasible solution from another feasible solution, *improvement* heuristics ( $I^*$ ) that try to improve a given feasible solution, and *consensus based* heuristics ( $CB^*$ ) that take sets of edges from two solutions of memory and construct a partial (infeasible) solution based on them. Finally, to avoid memory overload, after each construction, improvement or consensus based heuristic run, a destruction heuristic ( $D$ ) is used to remove a solution from  $M$ .

### 8.5.1 Global Data Structures

Some global variables are used by all heuristics. The matrix  $A$  stores the length of a shortest path between all pairs of nodes in  $G$ . The element  $A_{u,v}$  indicates the length of a shortest path between  $u$  and  $v$  in  $G$ , for all  $u, v \in V(G)$ . A shortest path is stored in the matrix  $\Pi = (\Pi_{u,v})$ . The element  $\Pi_{u,v}$  is the predecessor of  $v$  in some shortest path between  $u$  and  $v$  in  $G$ , for all  $u, v \in V(G)$ . The initial values of variables  $A_{u,v}$  and  $\Pi_{u,v}$  are calculated with the Floyd–Warshall’s shortest paths algorithm in time  $O(n^3)$  [41]. The variable *best\_sol* stores the solution found so far with lowest cost. The objective function of the solution with largest cost is stored in *worst\_sol*.

The shared memories  $M_i$ ,  $1 \leq i \leq m$  consist of finite linear lists of  $k_i$  solutions. The `read` operation does not take any input, and returns the solution stored in a position of  $M$  chosen at random. Initially,  $M$  is filled with connections obtained with the construction heuristic  $C1$ , as described below. On the other hand, the input of the `write` operation is a solution  $s$ . The algorithm `write` initially tests if  $f(s) < f(\text{worst\_connection})$  and  $s \neq s', \forall s' \in S$ . If this is false,  $s$  is discarded and the writing operation is aborted. Otherwise, the “roulette wheel” method is used to randomly select a position  $p_i$ ,  $1 \leq p_i \leq k_i$ , according to the value of the objective function for the solutions in  $M_i$ . Then, the solution  $s$  is assigned to  $M_i[k]$ .

### 8.5.2 Heuristic Strategies

In what follows, we describe simple heuristics used to generate or modify solutions to the PPC problem. These heuristics form the  $H$  set of agents operating on the shared memories. In the following description, we denote by  $u \rightsquigarrow v$  a path from vertex  $u$  to vertex  $v$ .

*Construction heuristic C1.* This heuristic creates a completely random solution.

- 1:  $D \leftarrow$  set of destinations;  $S \leftarrow$  set of sources
- 2:  $sol \leftarrow \emptyset$

```

3:  $i \leftarrow 1$ 
4: while  $D \neq \emptyset$  do
5:    $d \leftarrow$  random destination in  $D$ 
6:    $D \leftarrow D \setminus d$ 
7:    $path \leftarrow$  random path in  $G$  from  $S_i$  to  $d$ 
8:    $sol \leftarrow sol \cup path$ 
9:    $i \leftarrow i + 1$ 
10: end while
11: return  $sol$ 

```

*Time complexity:*  $O(pm)$

**Construction heuristic C2.** This heuristic takes a set  $E'$  of edges from the incomplete solution created by **CB1** or **CB2** (see below), and creates a complete solution as follows:

```

1:  $sol \leftarrow \emptyset$ 
2: for each edge  $(u, v) \in E'$  do
3:   Get a random source-destination pair  $(a, b)$ .
4:    $p_1 \leftarrow$  shortest-path( $a, u$ );  $p_2 \leftarrow$  shortest-path( $v, b$ )
5:    $p \leftarrow (p_1, p_2)$ 
6:    $sol \leftarrow sol \cup p$ 
7: end for
8: if  $|E'| < p$  then
9:   connect remaining source-destination pairs using C1.
9: end if

```

*Time complexity:*  $O(m)$

**Construction heuristic C3.** This heuristic constructs a solution  $s$  in the same way as **C1**. The difference is that it guarantees that the final solution has no sources or destinations inside each path  $(a, b) \in s$ . This makes the paths in  $s$  completely disjoint. Time complexity for **C3** is also  $O(pm)$ .

**Consensus based CB1.** Takes the common edges of two solutions and inserts them in an auxiliary memory. This algorithm works together with heuristic **C2** to form a new feasible solution.

```

1: Select solutions  $s_1$  and  $s_2$  from memory  $M$ .
2:  $E \leftarrow$  edges in  $s_1 \cup s_2$ 
3: store  $E$  in an auxiliary memory.

```

*Time complexity:*  $O(m \log m)$

**Consensus based CB2.** Compares two solutions  $s_1$  and  $s_2$  and stores the edges appearing only in the best of  $s_1, s_2$ .

```

1: Select solutions  $s_1$  and  $s_2$  from memory  $M$ .
2: if  $f(s_1) \geq f(s_2)$  then
3:    $s'_1 \leftarrow s_1$ ;  $s'_2 \leftarrow s_2$ 
4: else
5:    $s'_1 \leftarrow s_2$ ;  $s'_2 \leftarrow s_1$ 
6: end if

```

- 7: Let  $E \leftarrow \text{edges of } s'_1 \setminus s'_2$
- 8: store  $E$  in an auxiliary memory.

*Time complexity:*  $O(m \log m)$

**Improvement heuristic I1.** This heuristic uses the edge with lowest weight to concatenate shortest paths in a solution.

- 1: Select a solution  $s$  from memory  $M$ .
- 2:  $p \leftarrow a \rightsquigarrow b$ , for  $(a, b) \in s$ .
- 3: Let  $e \leftarrow (u, v)$  such that  $(u, v) \in p$  and  $(u, v)$  has minimum weight  $w(u, v)$ .
- 4:  $p_a \leftarrow \text{shortest-path}(a, u)$ ;  $p_b \leftarrow \text{shortest-path}(v, b)$
- 5:  $p' \leftarrow (p_a, u, v, p_b)$
- 6:  $s' \leftarrow (s \setminus p) \cup p'$
- 7: return  $s'$

*Time complexity:*  $O(m)$

**Improvement heuristic I2.** Replaces the weightiest edge from a path in a solution using a shortest-path.

- 1: Select a solution  $s$  from memory  $M$ .
- 2:  $p \leftarrow a \rightsquigarrow b$ , for  $(a, b) \in s$ .
- 3: Let  $e \leftarrow (u, v)$  such that  $(u, v) \in p$  and  $(u, v)$  has maximum weight  $w(u, v)$ .
- 4:  $p_a \leftarrow \text{shortest-path}(u, v)$ ;
- 5:  $p' \leftarrow (a \rightsquigarrow u, p_a, v \rightsquigarrow b)$
- 6:  $s' \leftarrow (s \setminus p) \cup p'$
- 7: return  $s'$

*Time complexity:*  $O(m)$

**Improvement heuristic I3.** Heuristic based on the “triangle inequality” property.

- 1: Select a solution  $s$  from memory  $M$ .
- 2: Let  $p \leftarrow a \rightsquigarrow b$  be the path in  $s$  between  $a$  and  $b$ , for  $(a, b) \in S \times D$ .
- 3: **for** each sub-path  $x, y, z$  in  $p$  **do**
- 4:   **if**  $\exists (x, z) \in E(G)$  such that  $w(x, z) < w(x, y) + w(y, z)$  **then**
- 5:      $p \leftarrow a \rightsquigarrow x, (x, z), z \rightsquigarrow b$
- 6:   **end if**
- 7: **end for**
- 8: return  $s'$

*Time complexity:*  $O(pm)$

**Improvement heuristic I4.** Creates a new solution from an existing solution, using only shortest paths.

- 1: Select a solution  $s$  from memory  $M$ 1.
- 2: **for** each source-destination pair  $(a, b) \in s$  **do**
- 3:    $p \leftarrow \text{shortest-path}(a, b)$
- 4:   Replace  $a \rightsquigarrow b$  by  $p$  in  $s$ .
- 5: **end for**
- 6: return  $s$

*Time complexity:*  $O(pm)$

*Destructor D.* Removes a solution from shared memory, using a stochastic criterion:

```

parameter Memory  $M_i$ 
 $s \leftarrow \text{roullet\_wheel}(M_i)$ 
delete( $s$ )

```

*Time complexity:*  $O(|M_i|) = O(k_i)$

## 8.6 A Partially Synchronous Distributed Model

The A-Team approach has a remarkable intrinsic parallelism: the application of various heuristics to the same state of the shared memory can be performed concurrently.

In implementing a parallel A-Team strategy, a problem that is faced is that the timing of heuristics is different. This can result in one heuristic being executed many times, while other heuristics have less opportunity to work in the parallel memory due to their greater complexity. This can lead to a situation where, although there is a team of heuristics, only a few of them is contributing to the final solution. On the other hand, one cannot completely synchronize the heuristics because this will make the whole algorithm depend on the slowest heuristic.

To avoid such timing problem, we use a partially synchronous model to implement the A-Team [44]. In such a model, one can fix a maximum level of asynchrony between tasks in different processors. These tasks are then coordinated by an external synchronizer. In this section, we define the formal characteristics of this model and describe the partial synchronous implementation of the A-Team.

### 8.6.1 The Partially Synchronous Model

We consider a distributed memory system composed by a set of  $p$  processors,  $p > 1$ , each one with its own memory. Like in a computer network, there is no shared global memory, and the processors interact only by message passing through (bi-directional) communication links. This parallel system is represented by an undirected connected graph  $H_{ps} = (Q, E(H_{ps}))$ , where  $Q$  represents the set of processors and  $E(H_{ps})$ , the communication links. For the remainder of the chapter, members of  $Q$  are referred to as *nodes*, whereas the members of  $E(H_{ps})$  are called *edges*. Every node  $q_i \in Q$  is able to communicate directly only with its neighbors in  $H_{ps}$ . The set of neighbors of  $q_i$  is denoted by  $N(i)$ .

The timing characteristics of the partially synchronous model are related to the following properties:

1. Each node  $q_i \in Q$  is driven by a local clock, which generates pulses  $s_i$  of duration  $\tau_i(s_i) \geq 0$ . During a pulse  $s_i$ , the node  $q_i$  can receive a set  $MSG_i(s_i)$

of messages from nodes in  $N(i)$  (this set is empty if  $\tau_i(s_i) = 0$ ), do some computation locally, and send a set of messages (possibly empty, and certainly empty if  $\tau_i(s_i) = 0$ ) to some nodes in  $N(i)$ . The computation performed by  $q_i$  during  $s_i$  changes the state of the local memory of  $q_i$  from  $\sigma_i(s_i)$  to  $\sigma_i(s_i + 1)$ .

The duration of a pulse is independent on the duration of other pulses, and can be arbitrarily large.

2. Let  $msg$  be a message sent by  $q_i$  in pulse  $s_i$  to a node  $q_j \in N(i)$ . If  $s_j$  is the pulse in  $q_j$  in which  $msg$  is received (i.e.,  $msg \in MSG_j(s_j)$ ), then  $s_j > s_i$  and  $s_j \leq s_i + \delta_{ij}$ , for some given constant  $\delta_{ij} \geq 1$ . Thus, the message delay of any communication is unpredictable, but bounded, no matter how long the duration of the pulses of the nodes involved in this communication are. Define  $\Delta_i = \max_{q_j \in N(i)} \delta_{ij}$ , for all  $q_i \in Q$ .

### 8.6.2 Distributed Memory Executions

The intrinsic parallelism in an A-Team is better characterized if we consider a parallel model with solutions shared across processors. In such a model, heuristics can execute concurrently and physically share the solutions in its local memory, making requests for solutions in other nodes of the system. In this case, the parallel reading operation returns a solution located at a specified position in  $M$ , and such position is selected at random. Still, the parallel writing operation takes a solution as input and must select an entry in  $M$  to store this solution. The entry is selected using a generalized “roulette wheel” method.

In our implementation, the memory  $M$  is evenly distributed over  $q$  processors, with  $M_i$  being the portion of the solution’s memory assigned to processor  $q_i$ . Each heuristic  $H_i \in H$  can run in any of the processors and access any of the other  $M_j \in M_{h_i}$  memories for reading or writing. The operation `read` and `write` must now take in consideration distributed issues arising from the concurrent access.

Let  $q_i \in Q$  be a processor which requests a distributed writing of a solution  $s$ . The first step is the choice of the specific memory  $M_i$  that will be written to. This can be made based on many characteristics of memories, or even randomly. The method used considers the *diversification* of each memory, where the diversification of a memory  $M_i$  is given by the value of the worst solution minus the value of the best solution in the processor. The choice is made for the memory with least diversification. Our aim is to contribute different solutions such that memories can share potential improvements.

To reduce communication costs, all reading and most of the writing operations are made in the local memory. This decision implies that writing solutions in remote processes is the only communication method between distinct memories. Therefore, we define for each process a parameter  $\alpha_w$ , the percentage of solutions that will be written to other (non local) memories.

Every node  $q_j \in N(q_i)$  must process requests for writing solutions in its local memory. It must select a position in its local memory independently, and this is done in the same way as with the sequential writing.

### 8.6.3 Types of Messages

The distributed operations on  $M$  involve some message exchange between processes in  $Q$ . These messages can be classified as initialization messages, writing messages, and control messages. For the purpose of synchronization, all messages include the pulse  $\sigma_t$  in which the distributed operation was requested to  $M_j$  by some heuristic  $H_i$ .

#### 8.6.3.1 Initialization Messages

The initialization messages are responsible for proper set up of variables. In the parallel A-Team, the initialization phase is started by a driver process which send a `startup` message to all child processes. The local variables are then initialized. An important substep in many of the proposed heuristics is the computation of the shortest path between two nodes. This is a time consuming task, and therefore all shortest paths are computed at the beginning of the A-Teams execution and stored in the matrices  $\Pi$  and  $A$  as explained in Sect. 8.5.1. To reduce the burden of calculating these values, the all-pairs of shortest paths algorithm is distributed over all processors. The distributed algorithm uses a torus model for diffusion of the computed values [16]. According to this model, values must be computed in two phases (corresponding to the two dimensions of the torus). The values in the first phase are sent using the `send_paths1` message. Then, the second phase of the algorithm proceeds and the result is sent using the `send_paths2` message. After this, the A-Team is initialized and start the processing of messages.

#### 8.6.3.2 Messages for Writing Solutions

The second type of messages is used when a process needs to write a solution. As explained, the first step consists in asking the neighbor nodes for a diversification parameter. Based on this parameter, the exact memory is chosen. The process then sends a message `ask_parameter` for each node in its neighborhood. Every time this message is received, the diversification parameter is computed and sent together with the `resp_parameter` message. Then, after processing `resp_parameter` from all neighbor nodes, a node can send the solution to the chosen node using the `receive_sol` message. The destination node receives the solution and inserts it in its local memory. Finally, after receiving all data the node sends the message `end_write` to the requesting node. This message acknowledges the correct reception of the solution.

### 8.6.3.3 Control Messages

The third kind of messages, control messages, are used to define the execution cycle of each process. There are two messages in this group: `work_done` and `terminate`. To understand the necessity of these messages, suppose that the *terminate condition* is true and the algorithm must terminate. How can all processors agree in stopping their operations? More importantly, how can a node be sure that no other node is depending on it for processing messages, and therefore it can stop safely? For this kind of coordination, the network must specify a life cycle control protocol.

The simple method employed in our implementation uses one of the nodes (the first one) as the arbitrator for execution termination. When the terminate condition is evaluated as true by the arbitrator (e.g., the optimum solution was found), it must send the message `terminate` to all other nodes. In response, the node send the same message as an acknowledgment. Each node that receives the `terminate` message must stop generating messages and only answer to requests previously sent to it. When the arbitrator receives an acknowledgment from all nodes it can safely send the `terminate` message. Finally, in response to the `terminate` message the process can shut down and exit. The processing of messages is summarized in Algorithm 2. Algorithm 3 shows the actions performed in each processor  $q_i$  during a pulse  $s_i$ .

### 8.6.4 Local Variables

Some local variables are used to handle the distributed operations on  $M$ . For example, the communication delays in the partially synchronous model allow a process  $q_i$  to have a number of pending distributed writings in a pulse. Then, a circular buffer *pending\_write<sub>i</sub>* is used to store, in processor  $q_i$ , the solutions associated with a pending writing requested by  $q_i$ . The size of this buffer must be equal to  $\delta_i$  because this is the maximum difference in pulses between  $q_i$  and any other process in the system. The extremes of the circular buffer are indicated by *first\_pending<sub>i</sub>*, which is the oldest solution in the circular buffer, and *last\_pending<sub>i</sub>*, which is the most recent solution in the circular buffer.

### 8.6.5 Simulating the Partially Synchronous Model

In this section, we deal with the problem of implementing the partially synchronous algorithm described in the previous section in an asynchronous model of distributed computation. The asynchrony in this model stems from the fact that all messages are delivered within some finite but unpredictable delay. In our case, the Parallel Virtual Machine (PVM) running in a local network is the underlying parallel system,

**Algorithm 2** Processing of messages in the parallel A-Team by node  $q_i$ **Message Processing**


---

```

1: Input: message message from node  $j$ 
2: switch message
3: {Initialization messages}
4: case startup:
5:   Compute shortest paths among nodes
     from  $(i - 1)(m/|Q|)$  to  $i(m/|Q|) - 1$ .
6:    $j \leftarrow i + m/|Q|$ 
7:   Send message send_paths1 to node  $q_j$ 
8: case send_paths1:
9:    $P \leftarrow$  path received with the message
10:  copy information in  $P$  to  $\Pi$  and  $A$ .
11:  send  $P$  to node  $q_{i+1}$ 
12: case send_paths2:
13:   $P \leftarrow$  path received with the message
14:  copy information in  $P$  to  $\Pi$  and  $A$ .
15: {Messages for writing solutions}
16: case ask_parameter:
17:   $p \leftarrow f(\text{best\_sol}) - f(\text{worst\_sol})$ 
18:  send message(resp_parameter) with parameter  $p$ 
19: case resp_parameter:
20:   $\text{resp}_j \leftarrow p$ , where  $p$  is the parameter received
21:  if  $\text{resp}_k \neq \emptyset$  for all  $k \neq i, k \in \{1, \dots, |M|\}$  then
22:     $\text{pos} \leftarrow k$  such that  $\text{resp}_k$  is minimum
23:     $\text{sol} \leftarrow \text{random}(\text{pending\_sol})$ 
24:     $\text{pending\_sol} \leftarrow \text{pending\_sol} \setminus \text{sol}$ 
25:    Send message receive_sol to node  $\text{pos}$ , with parameter  $\text{sol}$ .
26:  end if
27: case receive_sol:
28:   $\text{sol} \leftarrow$  solution encoded in the message
29:  call write(sol, M_i)
30:  send message end_write to node  $j$ 
31: case end_write:
32:   $\text{pending\_write} \leftarrow \text{pending\_write} - 1$ 
33: {Control messages}
34: case work_done:
35:   $\text{stop\_processing} \leftarrow \text{true}$ 
36: case terminate:
37:   $\text{terminate\_condition} \leftarrow \text{true}$ 
38: end case

```

---

therefore, delays can turn out to be a major bottleneck. The program must solve two main issues, namely keep the synchronization delay within the specified limits ( $\delta_i$ ) and implement a fair distribution of processing across nodes in the system.

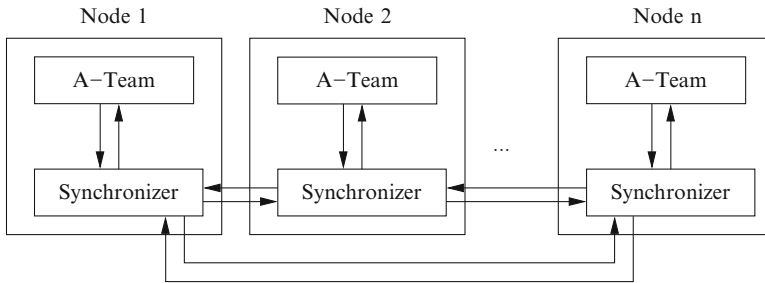
A simple way to solve this problem is to have, for each processing node of the A-Team, an auxiliary process which takes control of the communication requirements. Figure 8.3 represents graphically this architecture. Each node is now composed of two subtasks: the A-Team and a synchronizer module. All communication from and to the A-Team task is handled by the synchronizer. On the other

**Algorithm 3** Pseudo code for processing in node  $q_i$ **Message Processing for Node  $q_i$** 

```

1: receive startup message from synchronizer
2: while terminate_condition  $\neq$  true do
3:   receive safe( $s$ , MSG( $s$ ))
4:   Do message processing (Algorithm 3)
5:   if stop_processing  $\neq$  false then
6:      $h \leftarrow \text{random}(H)$ 
7:      $sol \leftarrow h(M_i)$ 
8:      $l \leftarrow \text{random}(|M|)$ 
9:     if  $l = 1$  then
10:      write( $sol$ ,  $M_i$ )
11:    else
12:      Send message ask_parameter for all nodes  $j$  such that  $j \in \{1, \dots, |M|\}$  and
         $j \neq i$ 
13:       $pending\_sol \leftarrow pending\_sol \cup sol$ 
14:    end if
15:  end if
16:  send(acknowledge)
17: end while

```

**Fig. 8.3** Organization of processes in the nodes of the parallel machine

hand, messages can be sent directly among synchronizers running in different nodes. This ensures that the messages are exchanged within the specified delays.

In this architecture, only a specified number of messages can be exchanged between the A-Team and its synchronizer. The following types of messages are used:

- *startup*: This message signals the start of the computation. In response to this message all initialization tasks are executed. The most important of these tasks is the distributed computation of the matrices  $A_{n \times n}$  and  $\Pi_{n \times n}$ , with all pairs of shortest paths, and the matrix of precedents in the shortest paths, respectively.
- *safe*: Sent by the synchronizer when it is safe to do new computations. This is an important message because all work of the A-Team is done as a response to *safe*.
- *acknowledge*: Sent by the A-Team process to acknowledge the receiving and processing of the *safe* message.

In addition to these types of messages, there is a specific way for the A-Team to send information to other nodes. To do this, the task must send a message to the synchronizer with an special tag, specifying the destination. This tag is just a number between 1 and  $|Q|$ . The A-Team task does not need to know the exact identifier of the other nodes, thus simplifying message processing. All messages received from other nodes are also handled by the synchronizer. The A-Team just needs to recover the node tag from each message. Each message (sent or received) must also include the current execution step  $s_i$ .

## 8.7 Computational Results

The asynchronous team, being an stochastic algorithm with no performance guarantee, needs to be tested to determine the overall performance for practical instances. Such performance must be tested against instances that have characteristics found on real world problems such as similar sparsity, appropriate cost distribution, and total number of members for multicast groups.

Therefore, to evaluate the proposed algorithms, we executed two types of experiments. The first experiment used a sequential implementation (i.e., number of processing nodes equal to 1). The second type of experiment tested the speedup due to the parallel implementation.

### 8.7.1 Hardware and Software Configuration

The parallel A-Team was implemented using the Parallel Virtual Memory System (PVM) [180]. The programming language used was the C language, with the GNU GCC compiler. The machine used was an IBM SP2 parallel system, with processors running at 66.7 MHz and 256 MB of memory in each node.

In the tests, we used random instances presented in Gomes et al. [81] to evaluate and compare the performance of our method.

The total size of memory for solutions used in the A-Teams is equal to the number of nodes  $n$ . During tests, this value showed good results for graphs with the size that we are considering. The value of  $\delta$  was set to 4 for each node  $q_i$ , according to some experimentation performed on the data.

### 8.7.2 Sequential Tests

The first experiment executed tried to assess the computational performance of the A-Team approach, compared to other methods. We did two kinds of comparisons: against both an exact method and an approximate algorithm. The exact method

**Table 8.1** Computational results comparing A-Team to a branch-and-cut algorithm for instances with 30 nodes

| Instance |          |          | Branch-and-cut |             | A-Team   |            |           |              |           |
|----------|----------|----------|----------------|-------------|----------|------------|-----------|--------------|-----------|
| <i>n</i> | <i>m</i> | <i>p</i> | Opt. sol.      | CPU time    | Best run | Suc. ratio | Ave. sol. | Average time | Dist. (%) |
| 30       | 60       | 8        | 57             | 28 s        | 57       | 10         | 57.00     | 32 s         |           |
| 30       | 60       | 7        | 41             | 14 s        | 41       | 10         | 41.00     | 39 s         |           |
| 30       | 60       | 6        | 38             | 15 s        | 38       | 10         | 38.00     | 12 s         |           |
| 30       | 60       | 5        | 39             | 24 s        | 39       | 10         | 39.00     | 05 min 08 s  |           |
| 30       | 100      | 8        | 29             | 01 min 51 s | 29       | 1          | 29.80     | 05 min 02 s  | 2.76      |
| 30       | 100      | 7        | 25             | 09 s        | 25       | 10         | 25.00     | 02 min 03 s  |           |
| 30       | 100      | 6        | 30             | 49 s        | 30       | 10         | 30.00     | 01 min 04 s  |           |
| 30       | 100      | 5        | 21             | 01 min 14 s | 22       | 0          | 24.70     | 03 min 33 s  | 17.62     |
| 30       | 200      | 8        | 21             | 02 min 50 s | 21       | 10         | 21.00     | 03 min 28 s  |           |
| 30       | 200      | 7        | 19             | 52 min 55 s | 21       | 0          | 21.50     | 15 min 24 s  | 13.16     |
| 30       | 200      | 6        | 18             | 17 s        | 18       | 10         | 18.00     | 23 s         |           |
| 30       | 200      | 5        | 22             | 42 s        | 22       | 10         | 22.00     | 06 s         |           |
| 30       | 300      | 8        | 13             | 19 min 31 s | 13       | 8          | 13.20     | 18 min 46 s  | 1.54      |
| 30       | 300      | 7        | 13             | 02 min 59 s | 13       | 10         | 13.00     | 01 min 46 s  |           |
| 30       | 300      | 6        | 11             | 04 min 14 s | 11       | 10         | 11.00     | 23 s         |           |
| 30       | 300      | 5        | 11             | 01 min 36 s | 11       | 10         | 11.00     | 25 s         |           |
| 30       | 400      | 8        | 14             | 48 s        | 14       | 10         | 14.00     | 09 min 36 s  |           |
| 30       | 400      | 7        | 14             | 59 min 43 s | 14       | 10         | 14.00     | 02 min 28 s  |           |
| 30       | 400      | 6        | 12             | 12 min 42 s | 12       | 10         | 12.00     | 02 min 30 s  |           |
| 30       | 400      | 5        | 10             | 01 min 19 s | 10       | 10         | 10.00     | 14 s         |           |

chosen was a branch-and-cut, implemented by Meneses [130]. Due to the hardness of the problem, the branch-and-cut algorithm could find exact results only for graphs with at most 40 nodes, 500 edges and  $p = 5$ .

For most instances, the A-Team found the optimum value in a fraction of the time spent by the exact algorithm. In Tables 8.1 and 8.2, we have a summary of results for the first experiment. The last column in this table represents the distance between the average solution found by the A-Team and the optimum value, when these values are different.

The second experiment was executed for instances with sizes ranging from 100 to 200 nodes, 500–2,000 edges and  $p = 10$ –20. For these graphs, we tried to solve the problem using both the A-Team and a Down Hill greedy heuristic. The results of these experiments appear in Table 8.3. We can see that the improvements in solution value are between 29 and 38%. The greater running time of the A-Teams can be explained by the fact that it needs to initialize memory, which increases with the size of the instance – something that is not required in the local search algorithm.

**Table 8.2** Computational results comparing A-Team to a branch-and-cut algorithm for instances with 40 nodes

| Instance |          |          | Branch-and-cut |             |          | A-Team     |           |              |           |
|----------|----------|----------|----------------|-------------|----------|------------|-----------|--------------|-----------|
| <i>n</i> | <i>m</i> | <i>p</i> | Opt. sol.      | CPU time    | Best run | Suc. ratio | Ave. sol. | Average time | Dist. (%) |
| 40       | 100      | 8        | 50             | 02 min 08 s | 50       | 4          | 50.70     | 14 min 18 s  | 1.40      |
| 40       | 100      | 7        | 48             | 01 min 37 s | 49       | 0          | 49.00     | 10 min 08 s  | 2.08      |
| 40       | 100      | 6        | 38             | 25 s        | 38       | 10         | 38.00     | 01 min 05 s  |           |
| 40       | 100      | 5        | 32             | 47 s        | 32       | 10         | 32.00     | 11 s         |           |
| 40       | 200      | 8        | 30             | 08 min 24 s | 30       | 1          | 31.10     | 14 min 24 s  | 3.67      |
| 40       | 200      | 7        | 19             | 32 s        | 19       | 10         | 19.00     | 1 min 26 s   |           |
| 40       | 200      | 6        | 23             | 59 s        | 23       | 4          | 23.60     | 08 min 22 s  | 2.61      |
| 40       | 200      | 5        | 16             | 07 s        | 16       | 10         | 16.00     | 58 s         |           |
| 40       | 300      | 8        | 20             | 01 min 48 s | 20       | 2          | 21.00     | 10 min 31 s  | 5.00      |
| 40       | 300      | 7        | 14             | 01 min 19 s | 14       | 10         | 14.00     | 07 min 01 s  |           |
| 40       | 300      | 6        | 14             | 01 min 27 s | 14       | 10         | 14.00     | 56 s         |           |
| 40       | 300      | 5        | 20             | 06 min 27 s | 20       | 10         | 20.00     | 01 min 15 s  |           |
| 40       | 400      | 8        | 22             | 39 min 35 s | 22       | 5          | 22.70     | 09 min 30 s  | 3.18      |
| 40       | 400      | 7        | 14             | 08 min 37 s | 14       | 10         | 14.00     | 48 s         |           |
| 40       | 400      | 6        | 17             | 02 min 17 s | 17       | 10         | 17.00     | 01 min 36 s  |           |
| 40       | 400      | 5        | 11             | 01 min 42 s | 11       | 10         | 11.00     | 38 s         |           |
| 40       | 500      | 8        | 17             | 04 min 07 s | 17       | 10         | 17.00     | 07 min 29 s  |           |
| 40       | 500      | 7        | 16             | 04 min 33 s | 16       | 10         | 16.00     | 01 min 14 s  |           |
| 40       | 500      | 6        | 14             | 03 min 46 s | 14       | 10         | 14.00     | 01 min 04 s  |           |
| 40       | 500      | 5        | 12             | 10 min 45 s | 12       | 10         | 12.00     | 01 min 16 s  |           |

### 8.7.3 Parallel Tests

Another area of testing for the asynchronous team described in this chapter is their performance as a parallel procedure. Since the implicit parallelism of the asynchronous team can be a source of additional performance for the algorithm, it is interesting to see how it can be tuned such that communication issues are avoided.

The premise of the parallel tests is: given the same A-Team configuration as in the sequential tests, what speed up can we have with the increase of nodes in the parallel system? With this goal in mind, we run the program for instances shown in Table 8.4.

The organization of the parallel processes was done as follows. Initially, a parent process is launched in the first node of the SP2 machine. The responsibility of this process is just to launch identical copies of the `ateam` program in each of the remaining nodes. The parameters passed to the child processes are the instance to be solved and the random seed used throughout the computation. Thereafter, each node starts executing the partial synchronizer, which controls the computation. The organization of processes in each node is described in Fig. 8.3.

**Table 8.3** Computational results comparing A-Team to a local search algorithm

| Instance |          |          | Local search |              |            | A-Team    |              |            |             |
|----------|----------|----------|--------------|--------------|------------|-----------|--------------|------------|-------------|
| <i>n</i> | <i>m</i> | <i>p</i> | Best sol.    | Average sol. | Aver. time | Best sol. | Average sol. | Aver. time | Improv. (%) |
| 100      | 500      | 10       | 77           | 93.2         | 1.5        | 59        | 63.3         | 8.8        | 32          |
| 100      | 600      | 10       | 56           | 69.4         | 1.5        | 42        | 43.6         | 10.5       | 37          |
| 100      | 700      | 10       | 66           | 74.2         | 1.2        | 43        | 46.8         | 11.3       | 37          |
| 120      | 800      | 12       | 80           | 87.4         | 2.4        | 50        | 54.1         | 16.9       | 38          |
| 120      | 900      | 12       | 73           | 91.3         | 2.1        | 60        | 64.3         | 17.5       | 30          |
| 120      | 1,000    | 12       | 56           | 63.6         | 2.2        | 41        | 43.1         | 18.5       | 32          |
| 140      | 1,000    | 14       | 82           | 98.2         | 3.1        | 61        | 65.4         | 29.6       | 33          |
| 140      | 1,100    | 14       | 65           | 81.1         | 3.8        | 56        | 57.5         | 31.7       | 29          |
| 140      | 1,200    | 14       | 65           | 74.9         | 3.3        | 47        | 50.6         | 31.9       | 32          |
| 160      | 1,300    | 16       | 108          | 121.6        | 4.8        | 78        | 80.7         | 54.9       | 34          |
| 160      | 1,400    | 16       | 97           | 112.3        | 4.8        | 70        | 77.7         | 59.6       | 31          |
| 160      | 1,500    | 16       | 89           | 97.1         | 4.9        | 55        | 64.2         | 60         | 34          |
| 180      | 1,600    | 18       | 101          | 115.3        | 6.4        | 73        | 80.3         | 93.7       | 30          |
| 180      | 1,700    | 18       | 113          | 122          | 6.5        | 75        | 79.4         | 93.2       | 35          |
| 180      | 1,800    | 18       | 99           | 110          | 6.5        | 69        | 73.4         | 98.4       | 33          |
| 200      | 1,800    | 20       | 119          | 133.6        | 8.7        | 87        | 91.4         | 131.3      | 32          |
| 200      | 1,900    | 20       | 114          | 126          | 8.7        | 79        | 84.6         | 133.7      | 33          |
| 200      | 2,000    | 20       | 112          | 122.5        | 8.6        | 77        | 83.0         | 151.2      | 32          |

**Table 8.4** Computational results comparing the speed up of a parallel A-Team with 2, 4, 8, and 16 nodes

| Instance |       |     | Average time per number of nodes |        |      |       |      |       |      |       |      |
|----------|-------|-----|----------------------------------|--------|------|-------|------|-------|------|-------|------|
| $n$      | $m$   | $p$ | 1                                | 2      | S    | 4     | S    | 8     | S    | 16    | S    |
| 100      | 500   | 10  | 8.8                              | 6.00   | 1.47 | 5.00  | 1.76 | 3.66  | 2.41 | 2.43  | 3.62 |
| 100      | 600   | 10  | 10.5                             | 7.50   | 1.40 | 6.00  | 1.75 | 4.40  | 2.39 | 2.91  | 3.60 |
| 100      | 700   | 10  | 11.3                             | 9.00   | 1.26 | 6.00  | 1.88 | 4.62  | 2.44 | 3.12  | 3.63 |
| 120      | 800   | 12  | 16.9                             | 12.30  | 1.37 | 9.00  | 1.88 | 6.96  | 2.43 | 4.57  | 3.70 |
| 120      | 900   | 12  | 17.5                             | 12.48  | 1.40 | 9.00  | 1.94 | 7.34  | 2.38 | 4.78  | 3.66 |
| 120      | 1,000 | 12  | 18.5                             | 12.50  | 1.48 | 9.98  | 1.85 | 7.74  | 2.39 | 5.01  | 3.69 |
| 140      | 1,000 | 14  | 29.6                             | 18.54  | 1.60 | 15.19 | 1.95 | 12.13 | 2.44 | 8.05  | 3.68 |
| 140      | 1,100 | 14  | 31.7                             | 20.98  | 1.51 | 16.91 | 1.87 | 13.17 | 2.41 | 8.76  | 3.62 |
| 140      | 1,200 | 14  | 31.9                             | 23.08  | 1.38 | 17.27 | 1.85 | 13.72 | 2.33 | 8.59  | 3.71 |
| 160      | 1,300 | 16  | 54.9                             | 36.27  | 1.51 | 28.98 | 1.89 | 22.91 | 2.40 | 14.93 | 3.68 |
| 160      | 1,400 | 16  | 59.6                             | 41.25  | 1.44 | 32.08 | 1.86 | 24.64 | 2.42 | 16.45 | 3.62 |
| 160      | 1,500 | 16  | 60                               | 39.80  | 1.51 | 32.24 | 1.86 | 25.21 | 2.38 | 16.36 | 3.67 |
| 180      | 1,600 | 18  | 93.7                             | 67.78  | 1.38 | 50.18 | 1.87 | 39.77 | 2.36 | 26.00 | 3.60 |
| 180      | 1,700 | 18  | 93.2                             | 65.96  | 1.41 | 47.98 | 1.94 | 40.05 | 2.33 | 24.79 | 3.76 |
| 180      | 1,800 | 18  | 98.4                             | 67.98  | 1.45 | 53.03 | 1.86 | 41.78 | 2.35 | 26.17 | 3.76 |
| 200      | 1,800 | 20  | 131.3                            | 91.37  | 1.44 | 69.06 | 1.90 | 53.84 | 2.44 | 35.52 | 3.70 |
| 200      | 1,900 | 20  | 133.7                            | 92.08  | 1.45 | 73.04 | 1.83 | 56.44 | 2.37 | 35.79 | 3.74 |
| 200      | 2,000 | 20  | 151.2                            | 100.03 | 1.51 | 81.22 | 1.86 | 63.55 | 2.38 | 41.69 | 3.63 |

The results obtained are summarized in Table 8.4. The last columns represent the average time required to find the same solutions of Table 8.3. The columns labeled *S* represent the speedup of the corresponding configuration. The speedup is defined as

$$S(n) = \frac{T(1)}{T(n)}.$$

In the preceding formula,  $T(1)$  is the time for the sequential program and  $T(n)$  is the time for the parallel program with  $n$  processing nodes.

From the analysis of results on Table 8.4, we can see that with the increase of the number of nodes, the speed up of the system also increases. This is due to a greater number of possible solutions which can be simultaneously generated, but there are also reductions in the time for initialization and for finding the solutions to the shortest path problem.

## 8.8 Concluding Remarks

In this chapter, we discussed the PPCP, a NP-Hard problem with applications in the areas of telecommunications and VLSI. We presented a parallel, randomized optimization algorithm for the PPCP and discussed how it can be implemented using a partially synchronous model.

Experimentation with the algorithm shows that it gives good results compared with an exact method (branch-and-cut) and another randomized technique (a down-hill algorithm). The experiments also suggest that the performance of the parallel algorithm improves as the number of nodes in the system increases.

The PPC problem is still not so well studied as other classical problems, and therefore there may be many opportunities for improving the existing solutions for this problem. The techniques presented here are also not yet in widespread use, but can be successfully applied in many other combinatorial optimization problems.



## Chapter 9

# Streaming Cache Placement

Multicast algorithms involve more than finding best routes connecting a set of nodes. To design cost effective networks, it is necessary to consider other important measures of cost and efficiency which are sometimes overlooked.

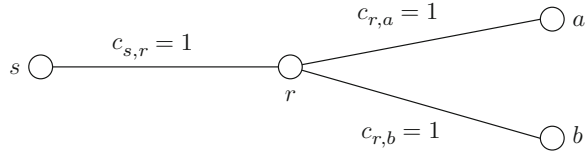
In this chapter, we investigate the construction of routing trees in which the cost of splitters is an important structural consideration. The resulting model adds information not only about the route followed by packets in the multicast tree, but also the number of locations of data is replicated. This makes it possible to minimize the costs associated with installing and maintaining multicast-enabled routers on a network.

### 9.1 Introduction

Multicast protocols are used to send information from one or more sources to a large number of destinations using a single send operation. Networks supporting multicast protocols have become increasingly important for many organizations due to the large number of applications of multicast, which include data distribution, video-conferencing [63], groupware [37], and automatic software updates [87]. Due to the lack of multicast support in existing networks, there is an arising need for updating unicast oriented networks. Thus, there is a clear economical impact in providing support for new multicast enabled applications.

In this chapter, we study a problem motivated by the economical planning of multicast network implementations. The streaming cache placement problem (SCPP) has the objective of minimizing costs associated with the implementation of multicast routers. This problem has only recently received attention [126, 144] and presents many interesting questions from the algorithmic and complexity theoretic point of view.

**Fig. 9.1** Simple example for the cache placement problem



### 9.1.1 Motivation

In multicast networks, nodes interested in a particular piece of data are called a multicast group. The main objective of such groups is to send data to destinations in the most efficient way, avoiding duplication of transmissions, and therefore saving bandwidth. With this aim, special purpose multicast protocols have been devised in the literature. Examples are the PIM [52] and core-based [14] distribution protocols. The basic operation in these routing protocols is to send data for a subset of nodes, duplicating the information only when necessary.

Network nodes that understand a multicast protocol are called cache nodes because they can send multiple copies of the received data. Other nodes simply act as intermediates in the multicast transmission. The main problem to be solved is deciding the route to be used by packets in such a network. One of the simplest strategies for generating multicast routes is to maintain a routing tree, linking all sources and destinations. A similar strategy, which may reduce the number of needed cache nodes, consists in determining a feasible flow from sources to destinations such that all destinations can be satisfied.

A main economical problem, however, is that not all nodes understand these multicast routing protocols. Moreover, upgrading all existing nodes can be expensive (or even impossible, when the whole network is not owned by the same company, as happens in the Internet).

Suppose an extreme situation, where no nodes have multicast capabilities. In this case, the only possible solution consists in sending a separate copy of the required data to each destination in the group. However, in this case instances can become quickly infeasible, as shown in Fig. 9.1. Here, all edges have capacity equal to one, and nodes  $a$  and  $b$  are destinations. In this example, a feasible solution is found when  $r$  becomes a cache node. Thus, it is interesting to determine the minimum number of cache nodes required to handle a specified amount of multicast traffic, subject to link capacity constraints. This is called the SCPP.

### 9.1.2 Formal Description of the SCPP

Suppose that a graph  $G = (V, E)$  is given, with a capacity function  $c : E \rightarrow \mathbb{Z}_+$ , a distinguished source node  $s \in V$  and a set of destination nodes  $D \subset V$ . It is required that data be sent from node  $s$  to each destination. Thus, we must determine

a set  $R$  of cache nodes, used to retransmit data when necessary, and the amount of information carried by each edge, which is represented by variables  $w_e \in \mathbb{R}_+$ , such that  $w_e \leq c_e$ , for  $e \in E$ . The objective of the SCPP is to find a set  $R$  of minimum size corresponding to a flow  $\{w_e | e \in E\}$ , such that for each node  $v \in D \cup R$  there is a unit flow from some node in  $\{s\} \cup R$  to  $v$ , and the capacity constraints  $w_e \leq c_e$ , for  $e \in E$ , are satisfied.

A characterization of the set of cache nodes can be given in terms of the surplus of data at each node  $v \in V$ . Suppose that the number of data units sent by node  $v$ , also called surplus, is given by variable  $b_v \in \mathbb{Z}$ . Note that the node  $s$  must send at least 1 unit of information, so  $b_s$  must be greater than 0. Each destination is required to receive a unit of data, so it has a negative surplus (requirement) of  $-1$ . Now, suppose that, due to capacity constraints, we need to establish  $v$  as a cache node. Then, the surplus at this node cannot be negative, since it is also sending forward the received data. If  $v$  is also a destination, then the minimum surplus is 0 (in this case it is receiving 1 unit and sending 1 unit); otherwise,  $b_v \geq 1$ . Thus, the set of cache nodes  $R \subseteq V \setminus \{s\}$  is the one such that  $b_v \geq 0$  and  $v \in D$  or  $b_v > 0$  and  $v \in V \setminus D \cup \{s\}$ , for all  $v \in R$ .

### 9.1.3 Related Work

Problems in multicast routing have been investigated by a large number of researchers in the last decade. The most studied problems relate to the design of routing tables with optimal cost. In these problems, given a set of sources and a set of destinations, the objective is to send data from sources to destinations with minimum cost.

In the case in which there are no additional constraints, this reduces to the Steiner tree problem on graphs [60]. In other words, it is required to find a tree linking all destinations to the source with minimum cost. Using this technique, the source and destinations are the required nodes, the remaining ones being the Steiner nodes. Many heuristic algorithms have been proposed for this kind of problem [38, 65, 110, 112, 116, 167, 175].

The problems above, however, consider that all nodes support a multicast protocol. This is not a realistic assumption on existing networks, since most routers do not support multicast by default. Thus, some sort of upgrade must be applied, in terms of software or even hardware, to deploy multicast applications. Despite this important application, only recently researchers have started to look at this kind of problem.

In Mao et al. [126], the SCPP is defined, in the context of Virtual Private Networks (VPNs). In this paper, the SCPP was proven to be NP-hard, using a reduction from the EXACT COVER BY 3-SETS problem, and a heuristic was proposed to solve some sample instances. However, the paper does not give details about possible versions of the problem, and proceeds directly to deriving local search heuristics.

Another related problem is the Cache Placement Problem [120]. Here, the objective is to place replicas of some static document on different points of a network to increase accessibility and also decrease the average access time by any client. The important difference between this problem and the SCPP is that [120] does not consider multicast transmissions. Also, there are no restriction on the capacity of links, and data is considered to be placed at the locations before the real operation of the network.

## 9.2 Types of Cache Placement Problems

In this section, we discuss two versions of the SCPP. In the tree streaming cache placement problem (TSCPP), the objective is to find a routing tree which minimizes the number of cache nodes needed to send data from a source to a set of destinations. We also discuss a modification of this problem where we try to find any feasible flow from source to destinations, minimizing the number of cache nodes. The problem is called the flow streaming cache placement problem (FSCPP).

### 9.2.1 The Tree Cache Placement Problem

Consider a weighted, capacitated network  $G(V, E)$  with a source node  $s$  and a routing tree  $T$  rooted on  $s$  and spanning all nodes in  $V$ . Let  $D$  be a subset of the nodes in  $V$  that have a demand for a data stream to be sent from node  $s$ . The stream follows the path defined by  $T$  from  $s$  to the demand nodes and takes  $B$  units of bandwidth on every edge that it traverses. For each demand node, a separate copy of the stream is sent. Edge capacities cannot be violated. Note that, depending on the network structure, an instance of this problem can easily become infeasible.

To handle this, we allow stream splitters, or caches, to be located at specific nodes in the network. A single copy of the stream is sent from  $s$  to a cache node  $r$  and from there multiple copies are sent down the tree. The optimization problem consists in finding a routing tree and to locate a minimum number of cache nodes.

Figure 9.2 shows an example for this problem. In this example, if nodes  $a$  and  $b$  each require a stream (with  $B = 1$ ) from  $s$ , and node  $r$  is not a cache node, then

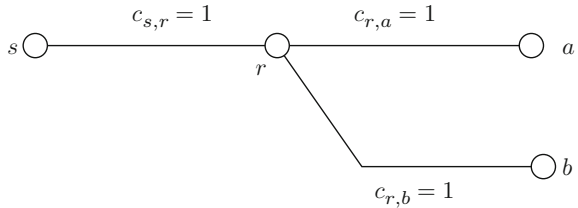


**Fig. 9.2** Simple example for the tree cache placement problem

we send 2 units from  $s$  to  $r$  and 1 unit from  $r$  to  $a$  and from  $r$  to  $b$ . We get an infeasibility on edge  $(s, r)$ , since 2 units flow on it, and it has capacity  $c_{s,r} = 1 < 2$ . However, if node  $r$  becomes a cache node, we can send 1 unit from  $s$  to  $r$  and then 1 unit from  $r$  to  $a$  and 1 unit from  $r$  to  $b$ . The resulting flow is now feasible.

To simplify the formulation of the problem, we can consider, without loss of generality, that the bandwidth used by each message is equal to 1.

The TSCPP is defined as follows. Given a graph  $G(V, E)$  with capacities  $c_{uv}$  on the edges, a source node  $s \in V$  and a subset  $D \subset V$  representing the destination nodes, we want to find a spanning tree  $T$  (which determines the paths followed by a data stream from  $s$  to  $v \in D$ ) such that the subset  $R \subseteq V \setminus \{s\}$ , which represents the cache nodes, has minimum size. For each node  $v \in D \cup R$ , there must be a data stream from some node  $w \in R \cup \{s\}$  to  $v$ , such that the sum of all streams in each edge  $(i, j) \in T$  does not exceed the edge capacity  $c_{ij}$ .

To state the problem more formally, consider an integer programming formulation for the TSCPP. Define the variables

$$y_e = \begin{cases} 1 & \text{if edge } e \text{ is in the spanning tree } T \\ 0 & \text{otherwise,} \end{cases}$$

$$x_i = \begin{cases} 1 & \text{if node } i \neq s \text{ is a cache node} \\ 0 & \text{otherwise} \end{cases}$$

$$b_i \in \{-1, \dots, |V| - 1\} \quad \text{the flow surplus for node } i \in V$$

$$w_e \in \{0, \dots, |V|\} \quad \text{the amount of flow in edge } e \in E.$$

Given the node-arc incidence matrix  $A$ , the problem can be stated as

$$\min \sum_{i=1}^{|V|} x_i \tag{9.1}$$

subject to

$$Aw = b \tag{9.2}$$

$$\sum_{i \in V} b_i = 0 \tag{9.3}$$

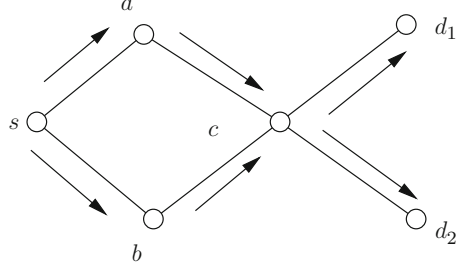
$$b_s \geq 1 \quad \text{for source } s \tag{9.4}$$

$$x_i - 1 \leq b_i \leq x_i(|V| - 1) - 1 \quad \text{for } i \in D \tag{9.5}$$

$$x_i \leq b_i \leq x_i(|V| - 1) \quad \text{for } i \in V - (D \cup \{s\}) \tag{9.6}$$

$$\sum_{e \in E} y_e = |V| - 1 \tag{9.7}$$

**Fig. 9.3** Simple example for the flow cache placement problem



$$\sum_{e \in G(H)} y_e \leq |H| - 1 \quad \text{for all } H \subset V \quad (9.8)$$

$$0 \leq w_e \leq c_e y_e \quad \text{for } e \in E \quad (9.9)$$

$$x \in \{0, 1\}^{|V|}, \quad y \in \{0, 1\}^{|E|} \quad (9.10)$$

$$b \in \mathbb{Z}, \quad w \in \mathbb{Z}^+, \quad (9.11)$$

where  $G(H)$  is the subgraph induced by the nodes in  $H$ . Constraint (9.2) imposes flow conservation. Constraints (9.3)–(9.6) require that there must be a number of data streams equal to the number of nodes in  $R \cup D$ . Constraints (9.7) and (9.8) are the spanning tree constraints. Finally, constraint (9.9) determine the bounds for flow variables, implying that the flow specified by  $w$  can be carried only on edges in the spanning tree.

### 9.2.2 The Flow Cache Placement Problem

An interesting extension of the TSCPP arises if we relax the constraints in the previous integer programming formulation that require the solution to be a tree of the graph  $G$ . Then, we have the more general case of a flow sent from the source node  $s$  to the set of destination nodes  $D$ . To see why this extension is interesting, consider the example graph, shown in Fig. 9.3. In this example, all edges have costs equal to 1. If we find a solution to the TSCPP on this graph, then a stream can be sent through only one of the two edges  $(s, a)$  or  $(s, b)$ . Suppose that we use edges  $(s, a)$  and  $(a, c)$ . This implies that  $c$  must be a cache node to satisfy demand nodes  $d_1$  and  $d_2$ . However, in practice, the number of caches in this optimal solution for the TSCPP can be further reduced.

Routing protocols, like OSPF, achieve load balancing by sending data through parallel links. In the case of Fig. 9.3, the protocol could just send another stream of data over edges  $(s, b)$  and  $(b, c)$ . If this happens, we do not need a cache node, and the solution will have fewer caches.

We define the FSCPP to be the problem of finding a feasible flow from source  $s$  to the set of destinations  $D$ , such that the number of required caches  $R \subseteq V \setminus \{s\}$  is minimized. The integer linear programming model for this problem is similar to (9.1)–(9.11), without the integer variable  $y$  and relaxing constraints (9.7) and (9.8).

### 9.3 Complexity of the Cache Placement Problems

We prove that both versions of the SSCP discussed above are NP-hard, using a transformation from SATISFIABILITY. This transformation allows us to give a proof of nonapproximation by showing that it is a gap-preserving transformation.

#### 9.3.1 Complexity of the TSCPP

In this section, we prove that the TSCPP is NP-hard, by using a reduction from SATISFIABILITY (SAT) [76].

**SAT:** Given a set of clauses  $C_1, \dots, C_m$ , where each clause is the disjunction of  $|C_i|$  literals (each literal is a variable  $x_j \in \{x_1, \dots, x_n\}$  or its negation  $\bar{x}_j$ ), is there a truth assignment for variables  $x_1, \dots, x_n$  such that all clauses are satisfied?

**Definition 1.** The TSCPP-D problem is the following. Given an instance of the TSCPP and an integer  $k$ , is there a solution such that the number of cache nodes needed is at most  $k$ ?

**Theorem 7.** *The TSCPP-D problem is NP-complete.*

*Proof.* This problem is clearly in NP, since for each instance  $I$  it is enough to give the spanning tree and the nodes in  $R$  to determine, in polynomial time, if this is a “yes” instance.

We reduce SAT to TSCPP-D. Given an instance  $I$  of SAT, composed of  $m$  clauses  $C_1, \dots, C_m$  and  $n$  variables  $x_1, \dots, x_n$ , we build a graph  $G(V, E)$ , with  $c_e = 1$  for all  $e \in E$ , and choose  $k = n$ . The set  $V$  is defined as

$$V = \{s\} \cup \{x_1, \dots, x_n\} \cup \{\bar{x}_1, \dots, \bar{x}_n\} \cup \{T'_1, \dots, T'_n\} \\ \cup \{T''_1, \dots, T''_n\} \cup \{T'''_1, \dots, T'''_n\} \cup \{C_1, \dots, C_m\},$$

and the set  $E$  is defined as

$$E = \bigcup_{i=1}^n \{(s, x_i), (s, \bar{x}_i)\} \cup \bigcup_{i=1}^n \{(x_i, T'_i), (\bar{x}_i, T'_i)\} \cup \bigcup_{i=1}^n \{(x_i, T''_i)\} \\ \cup \bigcup_{i=1}^n \{(\bar{x}_i, T'''_i)\} \cup \bigcup_{i=1}^m \left\{ \bigcup_{x_j \in C_i} (x_j, C_i) \bigcup_{\bar{x}_j \in C_i} (\bar{x}_j, C_i) \right\}. \quad (9.12)$$

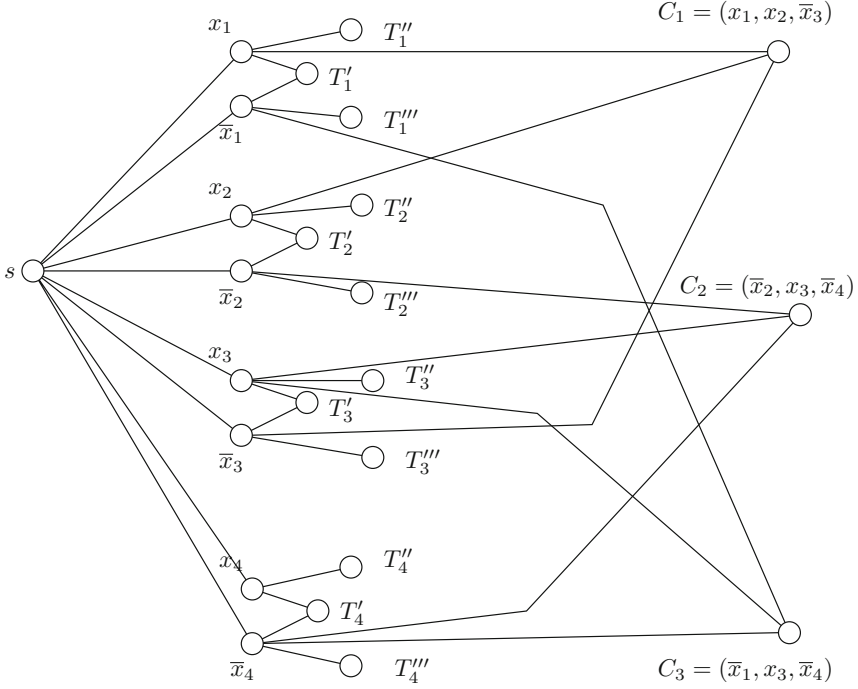


**Fig. 9.4** Small graph  $G$  created in the reduction given by Theorem 7. In this example, the SAT formula is  $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee x_3 \vee \bar{x}_4) \wedge (\bar{x}_1 \vee x_3 \vee \bar{x}_4)$

Figure 9.4 shows the construction of  $G$  for a small SAT instance. Define  $D = \{C_1, \dots, C_m\} \cup \{T'_1, \dots, T'_n\} \cup \{T''_1, \dots, T''_n\} \cup \{T'''_1, \dots, T'''_n\}$ . Clearly, destination nodes  $T'_i$ ,  $T''_i$  and  $T'''_i$  are there just to saturate the arcs leaving  $s$  and force one of  $x_i, \bar{x}_i$  to be chosen as a cache node. Also, each node  $C_i$  forces the existence of at least one cache among the nodes corresponding to literals appearing in clause  $C_i$ .

Suppose that the solution of the resulting TSCPP-D problem is true. Then, we assign variable  $x_i$  to true if node  $x_i$  is in  $R$ , otherwise we set  $x_i$  to false. This assignment is well defined, since exactly one of the nodes  $x_i, \bar{x}_i$  must be selected. Clearly, this truth assignment satisfies all clauses  $C_i$  because the demand of each node  $C_i$  is satisfied by at least one node corresponding to literals appearing in clause  $C_i$ .

Conversely, if there is a truth assignment  $\Gamma$  which makes the SAT formula satisfiable, we can use it to define the nodes which will be caches, and by construction of  $G$ , all demands will be satisfied. Finally, the resulting construction is polynomial in size, thus SAT reduces in polynomial time to TSCPP-D.  $\square$

**Fig. 9.5** Finding the optimal set of caches  $R$  for a fixed tree

**Find-Optimal-R**

```

1  Input: a tree  $T$ 
2  Output: a set  $R$  of cache nodes
3  for all  $v \in V$  do
4    if  $v \in D$  then
5       $demand(v) \leftarrow 1$ 
6    else
7       $demand(v) \leftarrow 0$ 
8    end
9  end call findR( $s$ )
10 return  $R$ 
11
12 procedure findR( $v$ )
13 for all  $w$  such that  $(v, w) \in T$  do
14   findR( $w$ )
15 end if  $v = s$  then
16   return  $R$ 
17 else
18    $p \leftarrow parent(v)$ 
19 end
20 if  $c_{p,v} < demand(v)$  then
21    $R \leftarrow R \cup \{v\}$ 
22    $demand(p) \leftarrow demand(p) + 1$ 
23 else
24    $demand(p) \leftarrow demand(p) + demand(v)$ 
25 end
26 end
27 end

```

As a simple consequence of this theorem, we have the following:

**Corollary 1.** *The TSCPP is NP-hard.*

It is interesting to observe that the problem remains NP-hard even for unity-capacity networks, since the proof remains the same for edges with unity capacity.

Some simple examples serve to illustrate the problem. For instance, if  $G$  is the complete graph  $K^n$ , then the optimal solution is simply a star graph with  $s$  at the center, and  $R = \emptyset$ . On the other hand, if the graph is a tree with  $n$  nodes, then the number of cache nodes is implied by the edges of the tree, thus the optimum is completely determined.

Figure 9.5 (Algorithm) determines an optimal set  $R$  from a given tree  $T$ . The algorithm works recursively. Initially, it finds the demand for all leaves of  $T$ . Then, it goes up the tree determining if the current nodes must be a cache node. The correctness of this method is proved below.

**Theorem 8.** *Given an instance of the TSCPP which is a tree  $T$ , then an optimal solution for  $T$  is given by Fig. 9.5 (Algorithm).*

*Proof.* The proof is by induction on the height  $h$  of a tree analyzed when Fig. 9.5 (Algorithm) arrives at line (1). If  $h = 0$ , then the number of cache nodes is clearly equal to 0. Assume that the theorem is true for trees with height  $h \geq 1$ . If the

capacity of the arc  $(p, v)$  is greater than the demand at  $v$ , then there is no need of a new cache node, and therefore the solution remains optimal. If, on the other hand,  $(p, v)$  does not have enough capacity to satisfy all demand at  $v$ , then we do not have a choice other than making  $v$  a cache node. Combining this with the assumption that the solution for all children of  $v$  is optimal, we conclude that the new solution for a tree of height  $h + 1$  is also optimal.  $\square$

### 9.3.2 Complexity of the FSCPP

We can use the transformation from SAT to TSCPP to show that FSCPP is also NP-hard. In the case of directed edges, this is simple, since given a graph  $G$  provided by the reduction, we can give an orientation of  $G$  from source to destinations. This is stated in the next theorem.

**Theorem 9.** *The FSCPP is NP-hard if the instance graph is directed.*

*Proof.* The proof is similar to the proof of Theorem 8. We need to make sure that the polynomial transformation given for the TSCPP-D also works for a decision version of the FSCPP. Given an instance of SAT, let  $G$  be the corresponding graph found by the reduction. We orient the edges of  $G$  from  $S$  to destinations  $D$ , that is, use the implicit orientation given in (9.12). It can be checked that in the resulting instance, the number of cache nodes cannot be reduced by sending additional flow in other edges other than the ones which form the tree in the solution of TSCPP. Thus, the resulting  $R$  is the same, and FSCPP is NP-hard in this case.  $\square$

Next, we prove a slightly modified theorem for the undirected version. To do this, we need the following variant of SAT:

3SAT(5): Given an instance of SATISFIABILITY with at most three literals per clause and such that each variable appears in at most five clauses, is there a truth assignment that makes all clauses true?

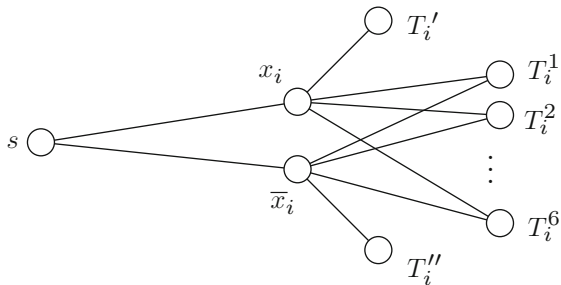
The 3SAT(5) is well known to be NP-complete [76].

**Theorem 10.** *The FSCPP is NP-hard if the instance graph is undirected.*

*Proof.* When the instance of FSCPP is undirected, the only thing that can go wrong is that some of the destinations  $T'_i$ ,  $T''_i$ , or  $T'''_i$  are being satisfied by flow coming from nodes  $C_j$  connected to their respective  $x_i$ ,  $\bar{x}_i$  nodes. What we need to do to prevent this is to bound the number of occurrences of each variable and add enough absorbing destinations to the subgraph corresponding to that variable. We do this by reduction from 3SAT(5). The reduction is essentially the same as the reduction from SAT to TSCPP, but now for each variable  $x_i$  we have nodes  $x_i$ ,  $\bar{x}_i$ ,  $T'_i$ ,  $T''_i$ , and  $T'''_i$ , for  $1 \leq k \leq 6$  (see Fig. 9.6). Also, for each variable  $x_i$  we have edges  $(s, x_i)$ ,  $(s, \bar{x}_i)$ ,  $(x_i, T'_i)$ ,  $(\bar{x}_i, T''_i)$ ,  $(x_i, T'''_i)$ ,  $(\bar{x}_i, T'''_i)$ , for  $1 \leq k \leq 6$ .

We claim that in this case for each pair of nodes  $x_i$ ,  $\bar{x}_i$ , one of them must be a cache node (which says that the corresponding variable in 3SAT(5) is true or

**Fig. 9.6** Part of the transformation used by the FSCPP



false). This is true because from the eight destinations not corresponding to clauses ( $T'_i$ ,  $T''_i$ , and  $T_i^k$ ,  $1 \leq k \leq 6$ ) attached to  $x_i$ ,  $\bar{x}_i$ , two can be directly satisfied from  $s$  without caches. However, the remaining six cannot be satisfied from nodes  $C_j$  linked to the current variable nodes because there are at most five such nodes. Thus, we must have one cache node at  $x_i$  or  $\bar{x}_i$ , for each variable  $x_i$ . It is clear that these are the only cache nodes needed to make all destinations satisfied. This gives us the correct truth assignment for the original 3SAT(5) instance. Conversely, any nonsatisfiable formula will transform to a FSCPP instance which needs more than  $n$  cache nodes to satisfy all destinations. Thus, the decision version of FSCPP is NP-complete, and this implies the theorem.  $\square$

Note that there is a case of TSCPP-D that is solvable in polynomial time, and this happens when  $k = 0$ , that is, determining if any cache node is needed. The solution is given by the following algorithm. Run the maximum flow algorithm from node  $s$  to all nodes in  $D$ . This can be accomplished, for example, by creating a dummy destination node  $d$  and linking all nodes  $v \in D$  to  $d$  by arcs with capacity equal to 1. If the maximum flow from  $s$  reaches each node in  $D$ , then the answer is true, since no cache node is needed to satisfy the destinations. Otherwise, the answer must be false because then at least one cache node is needed to satisfy all nodes in  $D$ .

## 9.4 Complexity of Approximation

As shown in the previous chapter, the SCPP in its two forms is NP-hard. We improve the hardness results for the SCPP, by showing that it is very difficult to give approximate solutions for such problems. General nonapproximation is proved using the reduction from SATISFIABILITY given in the previous chapter. Then, we improve the approximation results for the FSCPP using a reduction from SET COVER. In particular, given  $k$  destinations, we show that the FSCPP cannot have a  $\mathcal{O}(\log \log k - \delta)$ -approximation algorithm, for a very small  $\delta$ , unless NP can be solved in sub-exponential time.

In Sect. 9.5, we discuss the nonapproximation result for FSCPP based on the SATISFIABILITY problem. Then, in Sect. 9.6, we discuss the improved result for the FSCPP based on SET COVER.

## 9.5 Hardness of Approximation

The transformation used in Theorem 7 provides a method for proving a nonapproximation result for the TSCPP and FSCPP. We employ standard techniques, based on the gap-preserving transformations. To do this, we use an optimization version of 3SAT(5).

**MAX-3SAT(5):** Given an instance of 3SAT(5), find the maximum number of clauses that can be satisfied by any truth assignment.

**Definition 2.** For any  $\epsilon$ ,  $0 < \epsilon < 1$ , an approximation algorithm with guarantee  $\epsilon$  (or equivalently, an  $\epsilon$ -approximation algorithm) for a maximization problem  $\Pi$  is an algorithm  $A$  such that, for any instance  $I \in \Pi$ , the resulting cost  $A(I)$  of  $A$  applied to instance  $I$  satisfies  $\epsilon \cdot OPT(I) \leq A(I)$ , where we denote by  $OPT(I)$  the cost of the optimum solution. For minimization problems,  $A(I)$  must satisfy  $A(I) \leq \epsilon \cdot OPT(I)$ , for any fixed  $\epsilon > 1$ .

The following theorem from [7] is very useful to prove hardness of approximation results.

**Theorem 11 (Arora and Lund [7]).** *There is a polynomial time reduction from SAT to MAX-3SAT(5) which transforms formula  $\phi$  into a formula  $\phi'$  such that, for some fixed  $\epsilon$  ( $\epsilon$  is in fact determined in the proof of the theorem),*

- *If  $\phi$  is satisfiable, then  $OPT(\phi') = m$ , and*
- *If  $\phi$  is not satisfiable, then  $OPT(\phi') < (1 - \epsilon)m$ ,*

*where  $m$  is the number of clauses in  $\phi'$ .*

In the following theorem, we use this fact to show a hardness of approximation result for TSCPP.

**Theorem 12.** *The transformation used in the proof of Theorem 7 is a gap-preserving transformation from MAX-3SAT(5) to TSCPP. In other words, given an instance  $\phi$  of MAX-3SAT(5) with  $m$  clauses and  $n$  variables, we can find an instance  $I$  of TSCPP such that*

- *If  $OPT(\phi) = m$  then  $OPT(I) = n$ ; and*
- *If  $OPT(\phi) \leq (1 - \epsilon)m$  then  $OPT(I) \geq (1 + \epsilon_1)n$ .*

*where  $\epsilon$  is given in Theorem 11 and  $\epsilon_1 = \epsilon/15$ .*

*Proof.* Suppose that  $\phi$  is an instance of MAX-3SAT(5). Then, we can use the transformation given in the proof of Theorem 7 to construct a corresponding

instance  $I$  of TSCPP. If  $\phi$  has a solution with  $OPT(\phi) = m$ , where  $m$  is the number of clauses, then by Theorem 7, we can find a solution for  $I$  such that  $OPT(I) = n$ .

Now, if  $OPT(\phi) \leq (1 - \epsilon)m$ , then there are at least  $\epsilon m$  clauses unsatisfied. In the corresponding instance  $I$ , we have at least  $n$  cache nodes due to the constraints from nodes  $T'_i, T''_i$  and  $T'''_i$ ,  $1 \leq i \leq n$ . These cache nodes satisfy at most  $(1 - \epsilon)m$  destinations corresponding to clauses. Let  $U$  be the set of unsatisfied destinations. The nodes in  $U$  can be satisfied by setting one extra cache (in a total of two, for nodes  $x_j$  and  $\bar{x}_j$ ) for at least one variable  $x_j$  appearing in the clause corresponding to  $c_i$ , for all  $c_i \in U$ .

Thus, the number of extra cache nodes needed to satisfy  $U$  is at least  $|U|/5$ , since a variable can appear in at most 5 clauses. We have

$$OPT(I) \geq n + |U|/5 \geq n + \epsilon m/5 \geq (1 + \epsilon/15)n.$$

The last inequality follows from the trivial bound  $m \geq n/3$ . The theorem follows by setting  $\epsilon_1 = \epsilon/15$ .  $\square$

**Definition 3.** A Polynomial Time Approximation Scheme (PTAS) for a minimization problem  $\Pi$  is an algorithm that, for each  $\epsilon > 0$  and instance  $I \in \Pi$ , returns a solution  $A(I)$ , such that  $A(I) \leq (1 + \epsilon)OPT(I)$ , and  $A$  has running time polynomial in the size of  $I$ , depending on  $\epsilon$  (see, e.g., [147], page 425).

**Corollary 2.** Unless  $\mathcal{P} = \mathcal{NP}$ , the TSCPP cannot be approximated by  $(1 + \epsilon_2)$  for any  $\epsilon_2 \leq \epsilon_1$ , where  $\epsilon_1$  is given in Theorem 12, and therefore there is no PTAS for the TSCPP.

*Proof.* Given an instance  $\phi$  of SAT, we can use the transformation given in Theorem 11, coupled with the transformation given in the proof of Theorem 7, to give a new polynomial transformation  $\tau$  from SAT to TSCPP. Now, let  $I$  be the instance created by  $\tau$  on input  $\phi$ . Suppose there is an  $\epsilon_2$  approximation algorithm  $A$  for TSCPP with  $0 \leq \epsilon_2 \leq \epsilon_1$ . Then, when  $A$  runs on an instance  $I$  constructed by  $\tau$  from a satisfiable formula  $\phi$ , the result must have cost  $A(I) \leq (1 + \epsilon_2)n < (1 + \epsilon_1)n$ . Otherwise, if  $\phi$  is not satisfiable, then the result given by this algorithm must be greater than  $(1 + \epsilon_1)n$  because of the gap introduced by  $\tau$ . Thus, if there is an  $\epsilon_2$ -approximation algorithm, then we can decide in polynomial time if a formula  $\phi$  is satisfiable or not. Assuming  $\mathcal{P} \neq \mathcal{NP}$ , there is no such algorithm.

The fact that there is no PTAS for the TSCPP is a consequence of this result and the definition of PTAS.  $\square$

The above theorem and corollary can be easily extended to the FSCPP. The fact that the same transformation can be used for both problems can be used to demonstrate the nonapproximation result to the FSCPP as well. We state this as a corollary.

**Corollary 3.** Unless  $\mathcal{P} = \mathcal{NP}$ , the FSCPP has no PTAS.

*Proof.* The transformation from SAT to FSCPP is identical, so Theorem 12 is also valid for the FSCPP. This implies that the FSCPP has no PTAS, unless  $\mathcal{P} = \mathcal{NP}$ .  $\square$

## 9.6 Improved Hardness Result for FSCPP

In this section, we are interested in the case of general flows and directed arcs. This version of the problem is called the FSCPP. In particular, given  $k$  destinations, we show that the FSCPP cannot have a  $\mathcal{O}(\log \log k - \delta)$ -approximation algorithm, for a very small  $\delta$ , unless NP can be solved in sub-exponential time.

We have shown above that, given a instance of the FSCPP, there is an  $\epsilon > 0$  such that the FSCPP cannot be approximated by  $1 + \epsilon$ , thus demonstrating that FSCPP is MAX SNP-hard [145] and discarding the possibility of a PTAS. We show a stronger result: there is no approximation algorithm that can give a performance guarantee better than  $\log \log k$ , where  $k$  is the number of destinations. The proof is based on a reduction from the SET COVER problem.

**SET COVER:** Given a ground set  $T = t_1, \dots, t_n$ , with subsets  $S_1, \dots, S_m \subset T$ , find the minimum cardinality set  $C \subseteq \{1, \dots, m\}$  such that  $\bigcup_{i \in C} S_i = T$ .

It is known that SET COVER does not have approximation algorithms for any guarantee better than  $\mathcal{O}(\log n)$  [64]. Thus, if we find a transformation from SET COVER to FSCPP that preserves approximation, we can prove a similar result for FSCPP. We show how this transformation, which will be represented by  $\phi : \text{SC} \rightarrow \text{FSCPP}$ , can be done.

For each instance  $I_{\text{SC}}$  of set cover, we must find a corresponding instance  $I_{\text{FSCPP}}$  of the FSCPP. The instance  $I_{\text{SC}}$  is composed of sets  $T$  and  $S_1, \dots, S_m$  as shown above. The transformation consists of defining a capacitated graph  $G$  with a source and a set  $D$  of destinations. Let  $G$  be the graph composed of the following nodes:

$$V = \{s\} \cup \{w_1, \dots, w_m\} \cup \{v_1, \dots, v_n\} \cup \{s_1, \dots, s_m\}.$$

Also, let the edges  $E$  of the graph  $G$  be

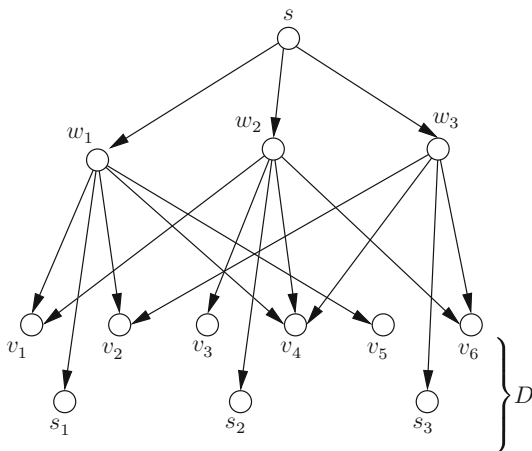
$$E = \{(w_j, v_i) \mid t_i \in S_j\} \cup \bigcup_{i=1}^m \{(s, w_i)\} \cup \bigcup_{i=1}^m \{(w_i, s_i)\}.$$

In the instance of FSCPP, the set of destination nodes  $D$  is given by

$$D = \{v_1, \dots, v_n\} \cup \{s_1, \dots, s_m\},$$

and  $s$  is the source node. Thus, there is an one to one correspondence between nodes  $w_i$  and sets  $S_i$ , for  $1 \leq i \leq m$ . There is also an one to one correspondence between nodes  $v_i$  and ground elements  $t_i \in T$ , for  $1 \leq i \leq n$ . There is a directed edge between the source and each node  $w_i$ , and between nodes  $w_i$  and nodes representing elements appearing in the set  $S_i$ . Nodes  $w_i$  are also linked to each  $s_i$ . Finally, each edge  $e$  has capacity  $c_e = 1$ . See an example of such reduction in Fig. 9.7. The ground set in this example is  $T = \{t_1, \dots, t_6\}$ , and the subsets are  $S_1 = \{t_1, t_2, t_4, t_5\}$ ,  $S_2 = \{t_1, t_2, t_4, t_6\}$ , and  $S_3 = \{t_2, t_4, t_6\}$ .

**Fig. 9.7** Example for transformation of Theorem 13



**Theorem 13.** *The transformation described above is a polynomial time reduction from SET COVER to FSCPP.*

*Proof.* Let  $I_{SC}$  be the instance of SET COVER and  $I_{FSCPP}$  the corresponding instance of the FSCPP. It is clear that the transformation is polynomial, since the number of edges and nodes is given by a constant multiple of the number of elements and sets in the instance of SET COVER. We must prove that  $I_{IS}$  and  $I_{SCPP}$  have equivalent optimal solutions. Let  $S'$  be an optimal solution for  $I_{SC}$ . First, we note that the destination nodes  $s_i$ ,  $1 \leq i \leq m$ , can be reached only from nodes  $w_i$ , and therefore each  $s_i$  must be satisfied with flow coming from  $w_i$ . Thus, each node  $s_i$  saturates the corresponding  $w_i$ , which means that to satisfy any other node from  $w_i$  we must make it a cache node. Then, we can clearly make  $R = \{w_i \mid i \in S'\}$ , and serve all remaining destinations in  $v_1, \dots, v_n$ , by definition of  $S'$ . Each node in  $R$  will be a cache node, and therefore  $R$  is a solution for  $I_{FSCPP}$ . This solution must be optimal because otherwise we could use a smaller solution  $R'$  to construct a corresponding set  $S'' \subset \{1, \dots, m\}$  with  $|S''| < |S'|$ , covering all elements of  $T$ , and therefore contradicting the fact that  $S'$  is an optimum solution for the SC instance. Thus, the two instances  $I_{SC}$  and  $I_{FSCPP}$  have equivalent optimal solutions.  $\square$

**Corollary 4.** *Given an instance  $I$  of SC, and the transformation  $\phi$  described above, then we have  $OPT(I) = OPT(\phi(I))$ .*

The following theorem, proved by Feige [64], will be useful for our main result.

**Theorem 14 (Feige [64]).** *If there is some  $\epsilon > 0$  such that a polynomial time algorithm can approximate set cover within  $(1 - \epsilon) \log n$ , then  $NP \subset TIME(n^{O(\log \log n)})$ .*

This theorem implies that finding approximate solutions with guarantee better than  $(1 - \epsilon) \log n$  for SET COVER is equivalent to solve any problem in NP in

sub-exponential time. It is strongly believed that this is not the case. We use this theorem and the reduction above to give a related bound for the approximation of FSCPP. To do this, we need a gap preserving transformation from SC to FSCPP, as stated in the following lemma.

**Lemma 3.** *If  $I$  is an instance SET COVER, then the transformation  $\phi$  from SC to FSCPP described above is gap preserving, that is, it has the following property:*

- (a) *If  $OPT(I) = k$  then  $OPT(\phi(I)) = k$ ; and*
- (b) *If  $OPT(I) \geq k \cdot \log n$  then  $OPT(\phi(I)) \geq k \cdot \log \log |D| - \delta$ ,*

*where  $k$  is a fixed value, depending on the instance, and*

$$\delta = -k \log \left\{ 1 - \log \left( 1 + \frac{n}{2^n} \right) / \log |D| \right\} \approx 0$$

*for large  $n$ .*

*Proof.* Part (a) is a simple consequence of Corollary 4. Now, for part (b), note that the maximum number of sets in an instance of SC with  $n$  elements is  $2^n$ . Consequently, in the instance of FSCPP created by transformation  $\phi$ ,  $|D| = m + n \leq 2^n + n$ . Thus, we have

$$\log |D| \leq \log(2^n + n) = n + \delta',$$

where  $\delta' = \log(1 + \frac{n}{2^n})$ . This implies that,

$$\log n \geq \log(\log |D| - \delta) = \log \log |D| + \delta'',$$

where  $\delta'' = \log(1 - \frac{\delta'}{\log |D|})$ . Therefore,

$$OPT(\phi(I)) \geq k \log n \geq k \log \log |D| - \delta,$$

where  $\delta = -k\delta''$  (note that  $\delta$  is a positive quantity). Finally, note that the quantity

$$-k \log \left\{ 1 - \log \left( 1 + \frac{n}{2^n} \right) / \log |D| \right\}$$

goes very fast to zero, in comparison to  $n$ , thus the value  $\log \log |D|$  is asymptotically optimal.

The reduction shown in Theorem 13 is gap preserving, since it maintains an approximation gap, introduced by the instances of SET COVER. Note, however, that the name “gap preserving” is misleading in this case, since the new transformation has a smaller gap than the original.

Finally, we get the following result.

**Theorem 15.** *If there is some  $\epsilon > 0$  such that a polynomial time algorithm  $A$  can approximate FSCPP within  $(1 - \epsilon) \log \log k$ , where  $k = |D|$ , then  $NP \subset TIME(n^{O(\log \log n)})$ .*

*Proof.* Suppose that an instance  $I$  of the SC is given. The transformation  $\phi$  described above can be used to find an instance  $\phi(I)$  of the FSCPP. Then,  $A$  can be used to solve the problem for instance  $\phi(I)$ . According to Lemma 3, transformation  $\phi$  reduces any gap of  $\log n$  to  $\log \log k$ . Thus, with such an algorithm, one can differentiate between instances  $I$  with a gap of  $\log n$ . But this is not possible in polynomial time, according to [64] unless  $NP \subset TIME(n^{O(\log \log n)})$ .

## 9.7 Conclusion

In this chapter, we presented and analyzed two combinatorial optimization problems occurring on multicast networks: TSCPP and its flow-based, generalized version, FSCPP [144]. We prove that both problems, on directed and undirected graphs, are NP-hard. For this purpose, we use a transformation from the SATISFIABILITY problem.

Many questions remain open for these problems. For example, it would be interesting to find algorithms with a better approximation guarantee, or improved nonapproximation results. Some of these issues will be considered in the next chapters.

We have shown that the SCPP in general cannot have approximation algorithms with guarantee better than  $\epsilon$ , for some  $\epsilon > 1$ . Thus, different from other optimization problems (such as the connected dominating set), the SCPP cannot have a PTAS.

We have also proved that the FSCPP cannot be approximated by less than  $\log \log k$ , where  $k$  is the number of destinations, unless NP can be solved in sub-exponential time. This shows that it is very difficult to find near optimal results for general instances of the FSCPP.



## Chapter 10

# Algorithms for Cache Placement

The results of the preceding chapter show that the SCPP is very difficult to solve, even if only approximate solutions are required. We describe some approximation algorithms that can be used to give solutions to the problem, and decrease the gap between known solutions and nonapproximation results. We also consider practical heuristics to find good near-optimal solutions to the problem. We propose two general types of heuristics, based on complementary techniques, which can be used to give good starting solutions for the SCPP [144].

### 10.1 Introduction

In this chapter, we present algorithms for solution of SCPP problems. Initially, we discuss algorithms with performance guarantee, also known as approximation algorithms. We give a general algorithm for SCPP problems, and also a better algorithm based on flow techniques. Approximation algorithms are very interesting as a way of understanding the complexity of the problem, but specially on this case, due to the negative results shown in Sect. 9.4, they are not very practical.

Thus, considering the complexity issues, we present a few polynomial time construction algorithms for the SCPP, based on two general techniques: adding destinations to a partial solution, and reducing the number of infeasible nodes in an initial solution. We report the results of computational experiments based on these two algorithms and its variations.

This chapter is organized as follows. In Sect. 10.2, we present algorithms with performance guarantee for the SCPP. In Sect. 10.3, we turn to algorithms without performance guarantee, and discuss a number of possible construction strategies. Then, in Sect. 10.4, we proceed to an empirical evaluation of the solutions returned by the proposed construction heuristics. Final remarks and future research directions are discussed in Sect. 10.5.

## 10.2 An Approximation Algorithm for SCPP

In this section, we present algorithms for the TSCPP and FSCPP and analyze their approximation guarantee. To simplify our results, we use the notation  $A(I) = |R \cup \{s\}|$ , where  $R$  is the set of cache nodes found by algorithm  $A$  applied to instance  $I$ . Also,  $OPT(I) = |R^* \cup \{s\}|$ , where  $R^*$  is an optimal set of cache nodes for instance  $I$ . Note that  $A(I) \geq 1$  and  $OPT(I) \geq 1$ , which makes Definition 2 valid for our problems.

### 10.2.1 A Simple Algorithm for TSCPP

It is easy to construct a simple approximation algorithm for any instance of the TSCPP. We denote by  $\delta_G(v)$  the degree of node  $v$  in the graph  $G$ .

Note that steps 3 and 4 of Fig. 10.1 (Algorithm) represent a worst case for Fig. 9.5 (Algorithm). The correctness of the algorithm is shown in the next lemma.

**Lemma 4.** *Figure 10.1 (Algorithm) returns a feasible solution to the TSCPP.*

*Proof.* The operation in step 2 maintains feasibility, since leaves cannot be used to reach destinations. The result  $R$  includes all internal nodes  $v$  with  $\delta_T(v) > 2$ , and all internal nodes  $v$  with  $\delta_T(v) = 2$  and  $v \in D$ . It suffices to prove that if  $\delta_T(v) = 2$  and  $v \notin D$  then  $v$  is not needed in  $R$ .

Suppose that  $v$  is an internal node with  $\delta_T(v) = 2$  and  $v \notin D$ . If the number of destinations down the tree from  $v$  is equal to 1, then  $v$  does not need to be a cache. Now, assume that the number of cache nodes down the tree from  $v$  is two or more. Then, there are two cases. In the first case, there is a node  $w$ , between  $v$  and the destinations, with  $\delta_T(w) > 2$ . In this case,  $w$  is in  $R$ , and we need to send 1 unit of flow from  $v$  to  $w$ , thus  $v$  does not need to be in  $R$ . In the second case, there must be some destination  $w$  with  $\delta_T(w) = 2$  between  $v$  and the other destinations. Again, in this case  $w$  will be included in  $R$  from  $S_2$ . Thus  $v$  does not need to be in  $R$ . This shows that  $R$  is a feasible solution to the TSCPP.  $\square$

**Lemma 5.** *Figure 10.1 (Algorithm) gives an approximation guarantee of  $|D|$ .*

#### Spanning Tree Algorithm

- 1 **Input:** Graph  $G$ , destinations  $D$ , source  $s$
- 2 **output:** a set  $R$  of cache nodes
- 3 **STEP1:** Construct a spanning tree  $T$  of  $G$
- 4 **STEP2:** Remove recursively all leaves of  $T$  which are not in  $D \cup \{s\}$
- 5 **STEP3:** Let  $S_1$  be the set of internal nodes  $v$  with  $\delta_T(v) > 2$
- 6 **STEP4:** Let  $S_2$  be the set of internal nodes  $v$  with  $\delta_T(v) = 2$  and  $v \in D$
- 7 **STEP5:** Return  $R = S_1 \cup S_2$ .

**Fig. 10.1** Spanning tree algorithm

*Proof.* Let us partition the set of destinations among  $D_1$  and  $D_2$ , where  $D_1 = D \setminus S_2$  and  $D_2 = S_2$ . Denote by  $D'$  the set of destinations which are leaves in  $T$ . Initially, note that for any tree the number of nodes  $v$  with degree  $\delta(v) > 2$  is at most  $|L| - 2$ , where  $L$  is the set of nodes with  $\delta(v) = 1$  (the leaves). But  $L$  in this case is  $D' \cup \{s\} \subseteq D_1 \cup \{s\}$ . This implies that  $|S_1| \leq |D_1 \cup \{s\}| - 2$ . Thus,

$$|R| = |S_1 \cup S_2| \leq |D_1 \cup \{s\}| - 2 + |D_2| = |D| - 1,$$

and

$$A(I) = |R| + 1 \leq |D| \leq |D|OPT(I),$$

since  $OPT(I) \geq 1$ . □

Let  $\Delta = \Delta(G)$  be the maximum degree of  $G$ . In the case in which all capacities  $c_e$ , for  $e \in E(G)$ , are equal to one, we can give a better analysis of the previous algorithm with an improved performance.

**Theorem 16.** *When  $c_e = 1$  for all  $e \in E$ , then Fig. 10.1 (Algorithm) is a  $k$ -approximation algorithm, where  $k = \min\{\Delta(G), |D|\}$ .*

*Proof.* The key idea to note is that if  $c_e = 1$  for all  $e \in E$ , then  $\lceil |D|/\Delta \rceil \leq OPT(I)$ , for any instance  $I$  of the TSCPP. This happens because each cache node (as well as the source) can serve at most  $\Delta$  destinations. Let  $A(I)$  be the value returned by Fig. 10.1 (Algorithm) on instance  $I$ . We know from the previous analysis of Lemma 5 that  $A(I) \leq |D|$ . Thus  $A(I) \leq \Delta OPT(I)$ . The theorem follows, since we know that this is also an  $|D|$ -approximation algorithm. □

### 10.2.2 A Flow-Based Algorithm for FSCPP

In this section, we present an approximation algorithm for the FSCPP. The algorithm is based on the idea of sending flow from the source to destination nodes. We show that this algorithm performs at least as good as the previous algorithm for the TSCPP. In addition, we show that for a special class of graphs this algorithm gives essentially the optimum solution. Therefore, for this class of graphs the FSCPP is solvable in polynomial time.

We give now standard definitions about network flows. For more details about the subject, see [3]. Let  $f(x, y) \in R^+$  be the amount of flow sent on edge  $(x, y)$ , for  $(x, y) \in E$ . A flow is called a feasible flow if it satisfies flow conservation constraints.

Let  $F(f, s, t) = \sum_{v \in V} f(s, v)$  be the total flow sent from node  $s$ . We assume that  $s$  can send at most  $\sum_{(s,v) \in E} c_{sv}$  units of flow, and  $t$  can receive at most  $\sum_{(u,t) \in E} c_{ut}$  units of flow. A feasible flow  $f$  is the maximum flow from  $s$  to  $t$  if there is no feasible flow  $f'$  such that  $F(f', s, t) > F(f, s, t)$ . A node  $v$  is a reached node from  $s$  by flow  $f$  if  $\sum_{w \in V} f(w, v) > 0$ . It is well known that when  $f$  is the maximum

**Flow Algorithm**

```

1   $Q \leftarrow \{s\}$ 
2  while  $D \setminus Q \neq \emptyset$  do
3    for all  $v \in Q$  do
4      find the maximum flow  $f_v$  from  $v$  to  $D \setminus Q$ 
5    end Let  $v^*$  be the node such that  $F(f_{v^*}, s, D)$  is maximum
6     $R \leftarrow R \cup \{v^*\}$ 
7    Add to  $Q$  the nodes reached by  $f_{v^*}$ 
8    for each edge  $(u, v) \in E$  do
9      Reduce capacity  $c_{u,v}$  by  $f_{v^*}(u, v)$ 
10   end
11 end

```

**Fig. 10.2** Flow algorithm

flow, then  $F(f, s, t) = C(s, t)$ , where  $C(s, t)$  represents the minimum capacity of any set of edges separating  $s$  from  $t$  in  $G$  (the minimum cut). We also use the notation  $C(U, \overline{U})$  to denote the total capacity of edges linking nodes in  $U$  to nodes in  $\overline{U}$ , where  $U \subset V$  and  $\overline{U} = V \setminus U$ .

Denote any feasible flow starting from node  $v$  by  $f_v$ . The algorithm works by finding the maximum flow from  $s$  to all nodes in  $D$ . If the total cost of this maximum flow is  $F(f_s, s, D) \geq |D|$ , then the problem is solved, since all destinations can be reached without cache nodes. Otherwise, we put all nodes reachable from  $s$  in a set  $Q$ . Then, we repeat the following steps until  $D \setminus Q$  is empty: for all nodes  $v \in Q$ , compute the maximum flow  $f_v$  from  $v$  to  $D$ . Find the node  $v^*$  such that  $f_{v^*}$  is maximum. Then, add  $v^*$  to  $R$  and add to  $Q$  all nodes reachable from  $v^*$ . Also, reduce the capacity of the edges in  $E$  by the amount of flow used by  $f_{v^*}$ . These steps are described more formally in Fig. 10.2 (Algorithm).

In the following theorem, we show that the running time of this algorithm is polynomial and depends on the time needed to find the maximum flow. Denote by  $C^M(G)$  the maximum value of the minimum cut between any pair of nodes  $v, w \in V(G)$ , that is,

$$C^M(G) = \max_{v, w \in V} C(v, w).$$

Similarly, we define

$$C^m(G) = \min_{v, w \in V} C(v, w).$$

**Theorem 17.** *Figure 10.2 (Algorithm) has running time  $\mathcal{O}(n \cdot |D| \cdot T^{mf} / C^m(G))$ , where  $T^{mf}$  is the time needed to run the maximum flow algorithm.*

*Proof.* The most costly operations in this algorithm are calls to the maximum flow algorithm. Therefore, we count the number of calls. Note that, at each of the  $N$  iterations of the while loop, a new element is added to the set of cache nodes. Thus,  $A(I)$  is equal to the number of such iterations.

Let  $v_i$  be the node added to the set of cache nodes at iteration  $i$ , and  $Q^i$  be the content of set  $Q$  at iteration  $i$  of Fig. 10.2 (Algorithm). At each step, the

number of elements of  $D$  found by the algorithm is, according to the maximum-flow/minimum-cut theorem, equal to the minimum cut from  $v_i$  to the remaining nodes in  $D \setminus Q^i$  (recall that all demands are unitary). Then,

$$|D| = \sum_{i=1}^N C(v_i, D \setminus Q^i) \geq \sum_{i=1}^N \min_{w \in D} C(v_i, w) \geq A(I) \min_{v, w \in V} C(v, w).$$

Thus, we have

$$N = A(I) \leq \frac{|D|}{C^m(G)}. \quad (10.1)$$

At each iteration of the while loop, the number of calls to the maximum flow algorithm is at most  $n$ . The total number  $n_c$  of such calls is given by  $n_c \leq n|D|/C^m(G)$ . Thus, the running time of Fig. 10.2 (Algorithm) is  $\mathcal{O}(n|D|T^{mf}/C^m(G))$ .  $\square$

## 10.3 Construction Algorithms for the SCPP

In this section, we provide construction algorithms for SCPP that give good results for a class of problem instances. The algorithms are based on dual methods for constructing solutions. In the first heuristic, the method used consists of sequentially adding destinations to the current flow, until all destinations are satisfied. The second method uses the idea of turning an initial infeasible solution into a feasible one, by adding cache nodes to the existing infeasible flow.

The general method used can be summarized by saying that it consists of the selection, at each step, of subsets of the resulting solution, while a complete solution is not found. To select elements of the solution, an ordering function is employed. This is done in a way such that parts of the solution which seem promising in terms of objective function are added first. In the remaining of this section, we describe the specific techniques proposed to create solutions for the SCPP.

### 10.3.1 Connecting Destinations

The first method we propose to construct solutions for the SCPP is based on adding destinations sequentially. The algorithm uses the fact that each feasible solution for the SCPP can be clearly described as the union of paths from the source  $s$  to the set  $D$  of destinations.

More formally, let  $D = \{d_1, \dots, d_k\}$  be the set of destinations. Then, a feasible flow for the SCPP is the union of a set of paths  $s = P_1 \cup \dots \cup P_k$ , such that  $P_1 = (s, x_1 \dots, x_{j_1-1}, x_{j_1})$ ,  $P_2 = (s, x_1 \dots, x_{j_2-1}, x_{j_2})$ ,  $\dots$   $P_k = (s, x_1 \dots, x_{j_k-1}, x_{j_k})$ , and where  $x_{j_i} = d_i$ , for  $i \in \{1, \dots, k\}$ .

**Construction Algorithm - I**

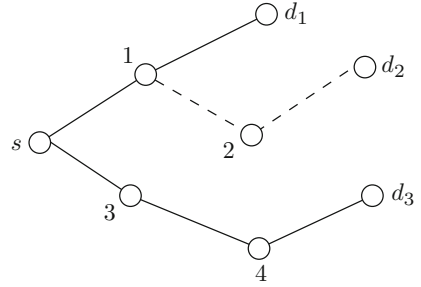
```

1  Input: graph  $G$ , set  $D$  of destinations
2   $A \leftarrow D$ 
3   $S \leftarrow \emptyset$   $R \leftarrow \emptyset$  while  $A \neq \emptyset$  do
4     $v \leftarrow \text{get\_dest}(A)$   $A \leftarrow A \setminus \{v\}$ 
5     $\mathcal{P} \leftarrow \text{get\_path}(v, R, G)$ 
6    if  $\mathcal{P} = \text{nil}$  then
7       $\mathcal{P} \leftarrow \text{get\_path}(v, S, G)$ 
8      Let  $w$  be the node connecting  $\mathcal{P}$  to  $S$ 
9       $R \leftarrow R \cup \{w\}$ 
10   end Remove from  $G$  the capacity used in  $\mathcal{P}$ 
11    $S \leftarrow S \cup \mathcal{P}$ 
12 end return  $R$ 

```

**Fig. 10.3** First construction algorithm for the SCPP

**Fig. 10.4** Sample execution for Fig. 10.3 (Algorithm). In this graph, all capacities are equal to 1. Destination  $d_2$  is being added to the partial solution, and node 1 must be added to  $R$



In the proposed algorithm, we try to construct a solution that is the union of paths. It is assumed initially that no destination has been connected to the source, and  $A$  is the set of nonconnected destinations, that is,  $A = D$ . Also, the set of cache nodes  $R$  is initially empty. During the algorithm execution,  $S$  represents the current subgraph of  $G$  connected to the source  $s$ . At each step of the algorithm, a path is created linking  $S$  to one of the destinations  $d \in A$ . First, the algorithm tries to find a path  $\mathcal{P}$  from  $d$  to one of the nodes in  $R \cup \{s\}$  (Fig. 10.4). If this is not possible (this is represented by saying that  $\mathcal{P} = \text{nil}$ ), then the algorithm tries to find a path  $\mathcal{P}$  from  $d$  to some node in the connected subgraph  $S$ . Let  $w$  be the connection point between  $\mathcal{P}$  and  $S$ . Then, add  $w$  to the set  $R$  of caches, since this is necessary to make to the solution feasible. Finally, the residual flow available in the graph is updated, to account for the capacity used by  $\mathcal{P}$ . The algorithm finishes when all destinations are included, and  $R$  is returned as the solution for the problem.

The formal steps of the proposed procedure are described in Fig. 10.3 (Algorithm). A number of important decisions, however, are unspecified in the description of the algorithm given so far. For example, there are many possible methods that can be used to select the next destination added to the current partial solution. Also, a path to a destination  $v$  can be found using diverse algorithms,

which can result in different selections for a required cache node. These possible variations in the algorithms are represented by two functions,  $\text{get\_path}(v, S)$ , and  $\text{get\_dest}(A)$ . Thus, changing the definition of these functions, we can actually achieve different implementations.

The first feature that can be changed, by defining a function  $\text{get\_dest}: 2^V \rightarrow V$ , is the order in which destinations are added to the final solution. Among the possible variations, we can list the following, which seem more useful:

- Give precedence to destinations closer to the source;
- Give precedence to destinations further from the source;
- Give precedence to destinations in lexicographic order.

The second basic decision that can be made is, once a destination  $v$  is selected, what type of path that will be used to join  $v$  to the rest of the graph. This decision is incorporated by the function  $\text{get\_path}: V \rightarrow 2^V$ . A second parameter involved in the definition of  $\text{get\_path}$  is the specific node  $w \in S$  which will be connected to  $v$ . Note that such a node always exist, since at each step of the construction there is a path from destinations to at least one node already reached by flow. Using a greedy strategy, the best way to link a new destination  $d$  is through a path from  $d$  to some node  $v \in R \cup \{s\}$ . However, it may not be possible to find such  $v$ , and this requires the addition of a new node to  $R$ . In both situations, it is not clear what is the optimal node to be linked to the current destination. Thus, another important decision in Fig. 10.3 (Algorithm) concerns how to choose, at each step, the node to be linked to the current destination.

### 10.3.1.1 Shortest Path Policy

Perhaps, the simplest and most logical solution to the above questions is to link destination nodes using shortest paths. This policy is useful, since it can be applied to answer the two questions raised above: the path is created as a shortest path, and the node  $v \in R \cup \{s\}$  selected is the closest from  $d$ . Thus, function  $\text{get\_path}(v, R, G)$  in Fig. 10.3 (Algorithm) becomes: select the node  $v \in R \cup \{s\}$  such that  $\text{dist}(v, d)$  (the shortest path distance between  $v$  and  $d$ ) is minimum; find the shortest path  $d \rightsquigarrow v$  from  $d$  to  $v$ ; add  $d \rightsquigarrow v$  to the current solution. If there is no path from  $d$  to  $R \cup \{s\}$ , then let  $v$  be the node closest to  $d$  and reached by flow from  $s$  and add  $v$  to  $R$ .

### 10.3.1.2 Other Policies

We have tested other two methods of connecting sources to destinations. In the first method, destinations are connected through a path found using the depth first search algorithm. In this implementation, paths are followed until a node already in  $R$ , or connected to some node in  $R$ , is found. In the last case, the connection node

must be added to  $R$ . The second method used employs random paths starting from the destination nodes. This method is just useful to understand how good are the previous algorithms compared to a random solution.

Once defined the policy to be used in the constructor, it is not difficult to prove the correctness, as well as finding the complexity of the algorithm.

**Theorem 18.** *Given an instance  $I$  of the SCPP, Fig. 10.3 (Algorithm) returns a correct solution.*

*Proof.* At each step, a new destination is linked to the source node. Thus, at the end of the algorithm all destinations are connected. The paths determined by such connections are valid, since they use some of the available capacity (according to the information stored in the residual graph  $G$ ). Nodes are added to  $R$  only when it is required to make the connection possible. Thus, the resulting set  $R$  is correct, and corresponds to a valid flow from  $s$  to the set  $D$  of destinations.

**Theorem 19.** *Using the closest node policy for destination selection and the shortest path policy for path creation, the time complexity of the Fig. 10.3 (Algorithm) is  $\mathcal{O}(|D|n^2)$ .*

*Proof.* The external loop is executed  $|D|$  times. The steps of highest complexity inside the while loop are exactly the ones corresponding to the `get_path` procedure. As we are proposing to use the shortest path algorithm for the implementation of `get_path`, the complexity is  $\mathcal{O}(n^2)$  (but can be improved with more clever implementations for the shortest path algorithm). Other operations have smaller complexity, thus the total complexity for Fig. 10.3 (Algorithm) is  $\mathcal{O}(|D|n^2)$ .

Other implementations of Fig. 10.3 (Algorithm) would result in a very similar analysis of complexity.

### 10.3.2 Adding Caches to a Solution

We propose a second general technique for creating feasible solutions for the SCPP. The algorithm consists of adding caches sequentially to an initial infeasible solution, until it becomes feasible. The steps are presented in Fig. 10.5 (Algorithm). At the beginning of the algorithm, the set of cache nodes  $R$  is empty, and a possibly infeasible subgraph, linking the source to all destinations gives the initial flow. Such an initial infeasible solution can be created easily with any spanning tree algorithm. In the description of our procedure, we define the set  $I$  of infeasible nodes to be the nodes  $v \in V \setminus \{s\}$  such that

$$\sum_{(w,v) \in E(G)} c(w,v) - \sum_{(v,w) \in E(G)} c(v,w) \neq b_v,$$

where  $b_v$  is the demand of  $v$ , which can be 0 or 1.

```

Construction Algorithm - II
1  Input: Graph  $G$ 
2   $R \leftarrow \emptyset$ 
3   $S \leftarrow \text{spanning\_tree}(G)$ 
4  Remove from  $G$  the capacity on edges used by  $S$ 
5   $I \leftarrow$  infeasible nodes in  $S$ 
6  while  $I \neq \emptyset$  (there are infeasible nodes) do
7       $v \leftarrow \text{select\_unfeasible\_node}$ 
8      Try to find different paths to satisfy  $v$ 
9      if a set  $\mathcal{P}$  of paths is found then
10         Remove from  $G$  the capacity used by  $\mathcal{P}$ 
11          $S \leftarrow S \cup \mathcal{P}$ 
12          $I \leftarrow I \setminus \{v\}$ 
13     else
14          $R \leftarrow R \cup \{v\}$ 
15     end
16 end return  $R$ 

```

Fig. 10.5 Second construction algorithm for the SCPP

```

Feasibility Test
1  [t] Input: current solution  $S$ , destinations set  $D$ 
2  for  $v \in D$  do
3       $b_v \leftarrow 1$ 
4  end  $I \leftarrow \emptyset$ 
5  for  $v \in V$  do
6       $\delta \leftarrow \sum_{(w,v) \in E(S)} c(w,v) - \sum_{(v,w) \in E(S)} c(v,w)$ 
7      if  $\delta \neq b_v$  then
8           $I \leftarrow I \cup \{v\}$ 
9      end
10 end return  $I$ 

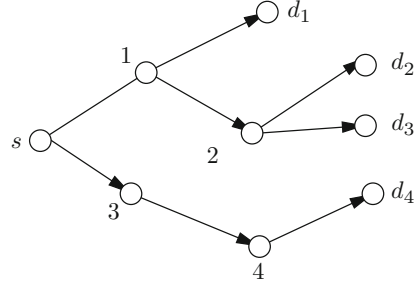
```

Fig. 10.6 Feasibility test for candidate solution

In the while loop of Fig. 10.5 (Algorithm), the current solution is initially checked for feasibility. This verification determines if there is any node  $v \in V$  such that the amount of flow leaving the node is greater than the arriving flow, or in other words,  $I \neq \emptyset$ . The formal description of this verification procedure is given in Fig. 10.6 (Algorithm). If the solution is found to be infeasible, then it is necessary to improve its feasibility by increasing the number of properly balanced nodes.

The correction of infeasible nodes  $v \in I$  is done in the body of the while loop in Fig. 10.5 (Algorithm). The procedure consists of selecting a node  $v$  from the set of infeasible nodes  $I$  in the flow graph, and trying to make it feasible by sending more data from one of the nodes  $w \in R \cup \{s\}$ . If this can be done in such a way that  $v$  becomes feasible again, then the algorithm just needs to update the current subgraph  $S$  and the set of infeasible nodes  $I$  (Fig. 10.7).

**Fig. 10.7** Sample execution for Fig. 10.5 (Algorithm), on a graph with unitary capacities. Nodes 1 and 2 are infeasible, and therefore are candidates to be included in  $R$



However, if  $v$  cannot receive enough additional data, then it must be added to the list of cache nodes. This clearly will make the node feasible again, since there will be no restrictions on the amount of flow departing from  $v$ . After adding  $v$  to  $R$ , the graph is modified as necessary. Example of possible modifications are changing the flow required by  $v$  to one unit and deleting additional paths leading to  $v$ , since only one is necessary to satisfy the flow requirements. We assume that these changes are done randomly, if necessary.

This construction technique can be seen as a dual of the algorithm presented in the previous subsection. In Fig. 10.3 (Algorithm), the assumption is that a partial solution can be incomplete, but always feasible with relation to the flow sent from  $s$  to the reached destinations. On the other hand, in Fig. 10.5 (Algorithm) a solution is always complete, in the sense that all destinations are reached. However, it does not represent a feasible flow, until the end of the algorithm.

Procedure `select_unfeasible_node` has the objective of finding the most suitable node to be processed in the current iteration. This is the main decision in the implementation of Fig. 10.5 (Algorithm), and can be done using a greedy function, which will find the best candidate according to some criterion. We propose some possible candidate functions, and determine empirically (in the next section) how these functions perform in practice.

- *Function* `largest_infeasibility`: select the node that has greatest infeasibility, that is, the difference between entering and leaving flow minus demand is maximum (breaking ties arbitrarily). This strategy tries to add to the set  $R$  a node which can benefit most from being a cache.
- *Function* `closest_from_source`: select the infeasible node which is closer to the destination. The advantage in this case is that a node  $v$  selected by this rule can help to reduce the infeasibility of other nodes down the path from  $s$  to the destinations.
- *Function* `uniform_random`: select uniformly a node  $v \in I$  to be added to  $R$ . This rule is useful for breaking the biases existing in the previous methods. It has also the advantage of being very simple to compute, and therefore very fast.

**Theorem 20.** *Given an instance of the SCPP, Fig. 10.5 (Algorithm) returns a correct solution.*

*Proof.* In the algorithm, the set of infeasible nodes  $I$  will decrease monotonically. This happens because at each step one infeasible node is selected and turned into a feasible node. Also, feasible nodes cannot become infeasible, since each operation requires that there is enough capacity in the network (this is guaranteed by the use of the residual graph  $G$ ). Thus, the algorithm terminates.

The data flow from the source to the destinations must be valid at the end, by definition of the set  $I$  (which must be empty at the end). Similarly, the set  $R$  must be valid, since it is used only to make nodes feasible in the case that no additional paths can be found to satisfy their requirements. Thus, the solution return by Fig. 10.5 (Algorithm) is correct for the SCPP.

**Theorem 21.** *The time complexity of the Fig. 10.5 (Algorithm) is  $\mathcal{O}(nmK)$ , where  $K$  is the sum of capacities in the SCPP instance.*

*Proof.* A spanning tree can be found in  $\mathcal{O}(m \log n)$ . Then, the next step is the while loop that will perform at most  $n$  iterations. The procedure `select_unfeasible_node` can be implemented in  $\mathcal{O}(n)$  by the use of some preprocessing, in each of the proposed implementations. Finding paths to infeasible nodes is clearly the most difficult operation in the loop. This can be performed in  $\mathcal{O}(m + n)$  for each path, using a procedure such as depth first search. However, it may be necessary to run this step a number of times proportional to the sum of capacities in the graph ( $K$ ), which results in  $\mathcal{O}(mK)$ . Other operations have low complexity, thus the maximum complexity for Fig. 10.5 (Algorithm) is  $\mathcal{O}(nmK)$ .

## 10.4 Experimental Results

In this section, we present computational experiments carried out with the construction algorithms proposed above.

All algorithms were implemented using the C programming language (the code is available by request). The resulting program was executed in a PC, with 312 MB of memory and a 800 MHz processor. The GNU GCC compiler (under the Linux operating system) was used with the optimization flag `-O2` enabled.

Table 10.1 presents a summary of the results of experiments with the proposed algorithms. The first two columns give the size of instances used. The remaining columns give results returned by Figs. 10.3 (Algorithm) and 10.5 (Algorithm) under different policies. Each entry reported in this table represents the averaged results over 30 instances of the stated size. Each instance was created using a random generator for connected graphs, first described in [80]. This generator creates graphs with random distances and capacities, but guarantees that the resulting instance is connected. Destinations were also defined randomly, with the number of destinations being equal to 40% of the size of the number of nodes. All instances assume that node 1 is the source node.

In columns 3–7 of Table 10.1, we show the comparison of results returned by different variations of Fig. 10.3 (Algorithm). The second and third columns give the

**Table 10.1** Computational results for different variations of Figs. 10.3 (Algorithm) and 10.5 (Algorithm). Times are given in seconds

| Instance |     | Constructor 1 |          |         |         |         | Constructor 2 |       |       |
|----------|-----|---------------|----------|---------|---------|---------|---------------|-------|-------|
| $n$      | $m$ | DFS           | Shortest | Random  |         |         | LI            | CS    | UR    |
| 50       | 500 | 9.9           | 2.9      | 18.8    | 18.3    | 18.4    | 3.0           | 2.8   | 3.2   |
| 50       | 800 | 12.9          | 4.9      | 29.3    | 29.6    | 28.4    | 5.0           | 4.8   | 5.2   |
| 60       | 500 | 35.1          | 8.8      | 56.8    | 56.4    | 55.5    | 9.6           | 9.2   | 9.9   |
| 60       | 800 | 44.7          | 11.7     | 77.6    | 76.9    | 76.8    | 12.5          | 11.7  | 12.8  |
| 70       | 500 | 78.9          | 17.0     | 111.3   | 111.2   | 110.4   | 18.9          | 18.1  | 19.6  |
| 70       | 800 | 102.2         | 20.3     | 139.3   | 140.1   | 139.7   | 22.5          | 21.4  | 23.3  |
| 80       | 500 | 147.6         | 27.8     | 180.7   | 182.2   | 181.3   | 32.0          | 31.1  | 33.4  |
| 80       | 800 | 186.1         | 32.4     | 218.3   | 218.8   | 218.9   | 36.9          | 36.2  | 38.9  |
| 90       | 500 | 239.8         | 42.1     | 268.4   | 267.8   | 268.5   | 49.5          | 49.9  | 52.0  |
| 90       | 800 | 285.4         | 47.5     | 312.6   | 313.4   | 313.3   | 56.5          | 57.4  | 59.8  |
| 100      | 500 | 344.1         | 60.3     | 368.0   | 369.0   | 369.1   | 71.3          | 74.2  | 75.8  |
| 100      | 800 | 398.6         | 67.7     | 420.1   | 421.6   | 422.3   | 80.3          | 84.3  | 85.4  |
| 110      | 500 | 466.4         | 84.2     | 482.6   | 484.8   | 485.8   | 100.3         | 106.1 | 106.2 |
| 110      | 800 | 525.5         | 92.5     | 541.7   | 545.1   | 543.7   | 111.9         | 119.5 | 118.4 |
| 120      | 500 | 599.7         | 115.1    | 611.3   | 614.9   | 613.3   | 137.6         | 146.9 | 144.9 |
| 120      | 800 | 668.7         | 125.5    | 676.9   | 680.6   | 679.2   | 151.9         | 164.0 | 159.4 |
| 130      | 500 | 748.8         | 157.4    | 751.4   | 755.6   | 754.9   | 180.3         | 194.6 | 189.0 |
| 130      | 800 | 824.7         | 169.5    | 823.3   | 827.7   | 827.8   | 199.3         | 215.7 | 208.5 |
| 140      | 500 | 910.6         | 213.2    | 906.2   | 909.8   | 908.9   | 233.4         | 252.4 | 243.9 |
| 140      | 800 | 995.9         | 228.2    | 984.8   | 990.2   | 989.3   | 254.6         | 276.3 | 265.5 |
| 150      | 500 | 1,092.0       | 281.8    | 1,074.4 | 1,080.3 | 1,080.6 | 292.0         | 319.6 | 303.6 |
| 150      | 800 | 1,185.4       | 299.6    | 1,162.6 | 1,170.0 | 1,169.1 | 315.8         | 347.0 | 353.6 |

solutions returned by the policies depth first search and shortest path, respectively. For both policies, it was not observed any significant change in behavior by using different orderings of the destinations. On the other hand, in the columns 5–7, we report the values returned by the random path policy using different ordering methods (closest, farthest from source, and lexicographic order, respectively). It seems that, given the weakness of the random path policy, the ordering method becomes a significant parameter in the determination of the solution quality.

Columns 8–10 of Table 10.1 present results for the execution of Fig. 10.5 (Algorithm). They correspond, respectively, to the largest infeasibility, closest from source, and uniform random policies. It is clear from the results that Fig. 10.5 (Algorithm) is very effective, in comparison with Fig. 10.3 (Algorithm). Although there is a tendency of getting better results with the “closest from source” policy, it seems that the output is relatively independent of the order of selection for infeasible nodes. Thus, it is probably better to use a simpler implementation, such as the “uniform random” policy.

Values for running time of the proposed algorithms are compared in Table 10.2. In this table, all values are given in milliseconds. As expected by the computational complexity results, Fig. 10.5 (Algorithm) has shown to spend more time than

**Table 10.2** Comparison of computational time for Figs. 10.3 (Algorithm) and 10.5 (Algorithm). All values are in milliseconds

| Instance |          | Constructor 1 |          |        |     |     | Constructor 2 |       |       |
|----------|----------|---------------|----------|--------|-----|-----|---------------|-------|-------|
| <i>n</i> | <i>m</i> | DFS           | Shortest | Random |     |     | LI            | CS    | UR    |
| 50       | 500      | 10            | 14       | 7      | 7   | 2   | 12            | 17    | 11    |
| 50       | 800      | 23            | 37       | 20     | 18  | 5   | 20            | 29    | 18    |
| 60       | 500      | 32            | 59       | 30     | 26  | 12  | 48            | 69    | 42    |
| 60       | 800      | 47            | 96       | 43     | 41  | 16  | 67            | 98    | 63    |
| 70       | 500      | 62            | 125      | 56     | 54  | 20  | 106           | 163   | 104   |
| 70       | 800      | 81            | 164      | 71     | 72  | 24  | 135           | 211   | 139   |
| 80       | 500      | 97            | 201      | 82     | 86  | 32  | 210           | 335   | 197   |
| 80       | 800      | 124           | 249      | 100    | 107 | 36  | 267           | 429   | 236   |
| 90       | 500      | 144           | 293      | 117    | 124 | 42  | 383           | 619   | 354   |
| 90       | 800      | 175           | 354      | 142    | 148 | 48  | 472           | 769   | 412   |
| 100      | 500      | 199           | 416      | 163    | 167 | 56  | 622           | 1,025 | 617   |
| 100      | 800      | 231           | 493      | 190    | 194 | 66  | 761           | 1,259 | 772   |
| 110      | 500      | 255           | 562      | 214    | 217 | 77  | 985           | 1,627 | 924   |
| 110      | 800      | 289           | 648      | 246    | 247 | 87  | 1,169         | 1,946 | 938   |
| 120      | 500      | 318           | 741      | 276    | 274 | 100 | 1,501         | 2,486 | 1,329 |
| 120      | 800      | 356           | 844      | 312    | 308 | 112 | 1,760         | 2,925 | 1,612 |
| 130      | 500      | 385           | 940      | 342    | 337 | 128 | 2,136         | 3,555 | 1,984 |
| 130      | 800      | 424           | 1,061    | 380    | 375 | 144 | 2,491         | 4,153 | 2,093 |
| 140      | 500      | 458           | 1,154    | 413    | 409 | 163 | 2,981         | 4,980 | 2,905 |
| 140      | 800      | 508           | 1,300    | 459    | 455 | 183 | 3,431         | 5,729 | 3,321 |
| 150      | 500      | 555           | 1,417    | 505    | 500 | 204 | 4,105         | 6,817 | 3,729 |
| 150      | 800      | 611           | 1,591    | 561    | 556 | 226 | 4,675         | 7,758 | 4,015 |

Fig. 10.3 (Algorithm). We note that, although these values can be improved by careful implementation, both algorithms have demonstrated good performance in practice (Fig. 10.8).

10.5 Conclusion

The SCPP is a difficult combinatorial optimization problem occurring in Multicast Networks. In this chapter, we described algorithms for solving the SCPP and some of its versions. Initially, we presented approximation algorithms for the SCPP. Although theoretically interesting, these algorithms cannot give good approximation guarantee in practice, due to the inherent problem complexity.

We also discussed how good solutions can be created by applying construction techniques. Two general types of techniques have being proposed, using dual points of view of the construction process. We described policies for implementation of these construction algorithms, and how different variants of the algorithms can be derived. Finally, we performed computational experiments to determine the quality of solutions generated by the different techniques described.

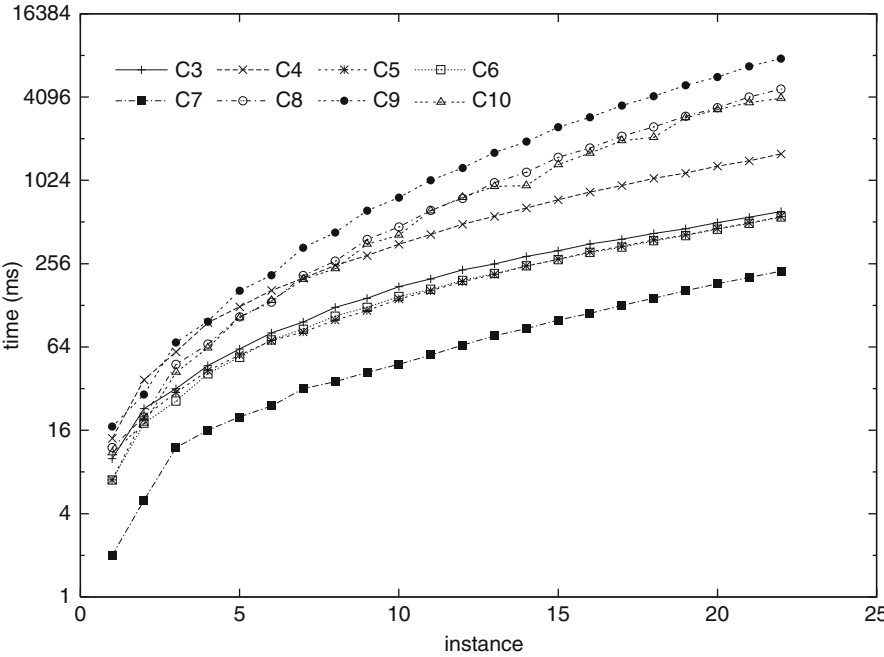


**Fig. 10.8** Comparison of computational time for different versions of Figs. 10.3 (Algorithm) and 10.5 (Algorithm). Labels “C3”–“C10” refer to the columns from 3 to 10 on Table 10.2

We believe that the techniques exposed here can be still further refined and improved. It would also be interesting to see these algorithms integrated with other heuristics. A future step is to use the proposed algorithms in the framework of general purpose metaheuristics, since this is the best way of achieving high quality solutions for most combinatorial problems. Clearly, other interesting open problems concern the proof of similar results for variants of the SCPP.

# Chapter 11

## Distributed Routing on Ad Hoc Networks

### 11.1 Introduction

In many applications of wireless networks, such as mobile commerce, search and rescue, and military battlefield operations, one has to deal with communication systems having no fixed topological structure. Such networks are usually referred to as ad hoc wireless systems.

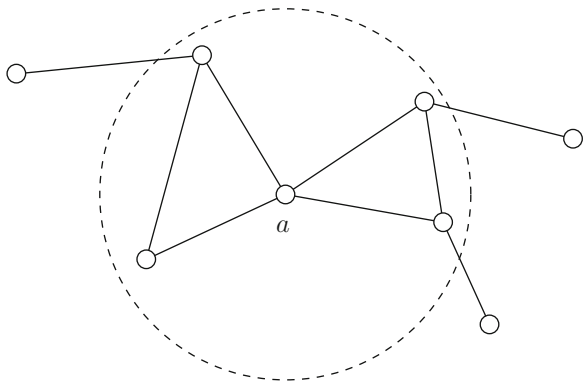
An essential problem concerning ad hoc wireless networks is to design routing protocols allowing for communication between mobile hosts. The dynamic nature of ad hoc networks makes this problem especially challenging. However, in some cases the problem of computing an acceptable virtual backbone can be reduced to the well-known minimum connected dominating set (MCDS) problem in unit-disk graphs [26, 27].

#### 11.1.1 Graph-Based Model of Ad Hoc Networks

Given a simple undirected graph  $G = (V, E)$  with the set of vertices  $V$  and the set of edges  $E$ , a dominating set (DS) is a set  $D \subseteq V$  such that each vertex in  $V \setminus D$  is adjacent to at least one vertex in  $D$ . If the graph is connected, a *connected dominating set* (CDS) is a DS which is also a connected subgraph of  $G$ . We note that computing the minimum CDS (MCDS) is equivalent to finding a spanning tree with the maximum number of leaves in  $G$ . In a *unit-disk* graph, two vertices are connected whenever the Euclidean distance between them is at most 1 unit.

Ad hoc networks can be modeled using unit-disk graphs as follows. The hosts in a wireless network are represented by vertices in the corresponding unit-disk graph, where the unit distance corresponds to the transmission range of a wireless device (see Fig. 11.1). It is known that both CDS and MCDS problems are NP-hard [76]. This remains the case even when they are restricted to planar, unit-disk graphs [10].

**Fig. 11.1** Approximating the virtual backbone with a connected dominating set in a unit-disk graph. The dashed line represents the transmission reach of transmitter  $a$



### 11.1.2 Existing Algorithms

Following the increased interest in wireless ad hoc networks, many approaches have been proposed for the MCDS problem in the recent years [6, 26, 47, 178]. Most of the existing heuristics are based on the idea of creating a dominating set incrementally, using a greedy algorithm. Some approaches try to construct a MCDS by finding a maximal independent set, which is then expanded to a CDS by adding “connecting” vertices [26, 178]. An independent set (IS) in  $G$  is a set  $I \subseteq V$  such that for each pair of vertices  $u, v \in I$ ,  $(u, v) \notin E$ . An independent set  $I$  is maximal IS if any vertex not in  $I$  has a neighbor in  $I$ . Obviously, any maximal independent set is also a dominating set.

There are several polynomial-time approximation algorithms for the MCDS problem. For instance, Guha and Khuller [84] propose an algorithm with approximation factor of  $H(\Delta) + 2$ , where  $\Delta$  is the maximum degree of the graph and  $H(n) = 1 + 1/2 + \dots + 1/n$  is the harmonic function. Other approximation algorithms are given in [26, 127]. A polynomial time approximation scheme (PTAS) for MCDS in unit-disk graphs is also possible, as shown by Hunt III et al. [92] and more recently by Cheng et al. [36].

A common feature of the currently available techniques for solving the MCDS problem is that the algorithms create the CDS from scratch, adding at each iteration some vertices according to a greedy criterion. For the general dominating set problem, the only exception known to the authors is briefly explained in [170], where a solution is created by sequentially removing vertices. A shared disadvantage of such algorithms is that they may require additional setup time, which is needed to construct a CDS from scratch. Another weakness of the existing approaches is that frequently they use complicated strategies to achieve a good performance guarantee.

In this chapter, we present a heuristic algorithm for computing approximate solutions to the MCDS problem. In particular, we discuss in detail the application of this algorithm to the MCDS problem in unit-disk graphs. The algorithm starts with a feasible solution, and recursively removes vertices from this solution, until a

minimal CDS is found (here, by a *minimal* CDS we mean a connected dominating set, in which removing any vertex would result in a disconnected induced subgraph). Using this technique, the proposed algorithm maintains a feasible solution at any stage of its execution; therefore, there are no setup time requirements. The approach also has the advantage of being simple to implement, with experimental results comparable to the best existing algorithms.

This chapter uses standard graph-theoretical notations. Given a graph  $G = (V, E)$ , a subgraph of  $G$  induced by the set of vertices  $S$  is represented by  $G[S]$ . The set of adjacent vertices (also called neighbors) of  $v \in V$  is denoted by  $N(v)$ . Also, we use  $\delta(v)$  to denote the number of vertices adjacent to  $v$ , i.e.,  $\delta(v) = |N(v)|$ .

The chapter is organized as follows. In Sect. 11.4, we present the algorithm and prove some results about its time complexity. In Sect. 11.4.1, we discuss a distributed implementation of this algorithm. Results of computational experiments with the proposed approach are presented in Sect. 11.5.2. Finally, in Sect. 11.6, we give some concluding remarks.

## 11.2 Ad Hoc Technology and Concepts

Ad hoc means “for this,” “for this purpose only.” Ad hoc networks are also referred to as Mobile Ad Hoc Networks (MANETs). The main feature of ad hoc networks is that each node acts as a client and a server unit. Ad hoc networks have a dynamic topology: no fixed infrastructure is required for their operation. The objectives of such ad hoc systems are generally: easy access, mobility, and easy disconnection and response to failures. Ad hoc systems commonly use what is called multi-hop routing, since each host can act as a router.

Several application scenarios can justify the use of ad hoc network systems. For example, we have the introduction of new mobile services. Another scenario involves the exploration of new geographical areas, as well as other applications such as rescue and military missions. New enabling technologies for ad hoc networks include point-to-point protocols, wireless systems, and other general types of ad hoc networks.

In general, the applications of ad hoc networks involve any situation where communication within a geographical region is required, no fixed communication system exists, and users are supposed to be in the reach of each other. Some additional examples include rescue groups, medical teams, and mobile systems for researchers in a conference. Finally, an important reason why ad hoc networks are important today is that they form the building block for the deployment of sensor networks, that is, networks that allow the integration of diverse activities, related to measurement, detection, and control, into real time sensors integrated in a wireless system. Applications of sensor networks have been envisioned in areas such as quality control, agriculture, defense, and social studies.

Ad hoc networks, however, present several challenges that must be addressed by researchers. Among them are issues such as limited resources (battery power, CPU),

limited security, due to extended exposition and increased vulnerability. Another problem is related to the difficulty in adapting existing protocols to the dynamic operation of such networks.

Several optimization problems are of importance in the design and management of ad hoc networks, for example, power control optimization [142], and minimization of errors in measurement [192]. In this chapter we are going to explore two problems: (1) computing an efficient backbone for routing on wireless systems and (2) maximizing total communication.

### 11.2.1 *Virtual Backbone Computation*

Virtual backbone computation is a method to improve the exchange of routing information in ad hoc system. The existing protocols for routing information exchange, having been developed for traditional systems, have several problems when adapted to wireless networks. One of the issues is related to information handling: most protocols in static networks are table-driven, that is, information is stored in tables that can be later used to retrieve routing directions. This method contrasts with the dynamic requirements of ad hoc networks. Moreover, protocols for static networks usually store too much information about the system, and due to the frequent changes in ad hoc networks, this information becomes outdated with higher frequency. Another reason for the inability of current protocols to perform well in ad networks is the need to spend bandwidth to track changes of positions of nodes.

The problems with existing protocols cited above have several consequences. First, this results in unneeded redundancy and large amount of collisions. It becomes necessary to use many packets to track changes in the network, and this results in higher collision rates. This pattern clearly increases the possibility of interference of routing traffic with normal traffic. The end result is, therefore, degradation of network throughput, and diminished bandwidth for the use of network applications.

Several solution techniques have been proposed in the last few years to avoid the problems described. One of the main techniques consist of employing a virtual backbone to reduce the amount of routing information shared among nodes of the system. A *virtual backbone* is a set of nodes that can be used to take routing decisions and that act as proxies for routing of packets. The objective of the technique is to reduce the amount of information that must be kept about nodes in the network, by requiring only a subset of nodes to be required for routing purposes.

Implementing a virtual backbone can be done by defining a small subset of nodes with the following property. (Dominating set property) Given a graph  $G = (V, E)$ , a subset  $V' \subseteq V$  has the dominating set property if each node  $v \in V$  is either an element of the dominating set, or adjacent to at least one element of  $V'$ . Thus, using a dominating set  $V'$  as a virtual backbone, the only information that must be shared for routing purposes is the location of elements of  $V'$ .

A formal definitions can be provided as follows. Let  $G = (V, E)$  be a simple undirected graph. A subset  $D \subseteq V$  is called a *dominating set* if any vertex in  $V$

either belongs to  $D$  or has a neighbor in  $D$ . A dominating set is said to be *minimum* if there is no smaller dominating set in the graph. A dominating set  $D$  is said to be *minimal* if there is no smaller dominating set contained in  $D$ .

### 11.2.2 Unit Disk Graphs

Ad hoc networks are frequently modeled as *unit disk graphs*. Such graphs have the properties of being planar, and edge have distance at most equal to 1. Planar graphs are the ones that represent nodes in an Euclidean space of two dimensions. In this case, it is equivalent to say that each node links to any other node within 1 unit of distance. For our purposes, arcs are considered to be undirected. An example of unit disk graph is shown in the figure below. In this example, dotted cycles represent the area covered by each wireless unit.

The main approach used to compute virtual backbones on wireless network systems is to approximating them using MCDSs on unit disk graphs.

## 11.3 Maximum Connected Dominating Sets

Algorithms for the MCDS are usually two phase algorithms. They consist of constructing an initial structure (e.g., an independent set), and subsequently making this structure to have the two desired properties: dominance and connectedness. Although effective, this strategy also has some associated problems. First, it requires a set up phase, where no transmissions are possible, since the backbone is nonexistent. Second, for the reason explained, this kind of algorithm can be difficult to reconfigure in practice.

A different type of algorithm can be proposed for the MCDS, which is composed of only one global phase. In the algorithm we describe next, there are at most  $n$  steps, and each step is composed alternatively by one of two decisions: selecting a node to become part of the MCDS, or remove it. The first type of step is called a *selection step*, the second type is called a *removal step*. If a decision is made to remove a node, then it is necessary to select a new neighbor to become part of the solution. The main feature of the described algorithm is that it starts with a feasible solution, composed of all nodes, and operates by removing nodes and improving the connected DS.

The selection of nodes for the MCDS is done according to the connectivity of nodes. At a selection step, a node with high connectivity is selected to become part of the final connected DS. If a node is a neighbor of a selected node, we try to remove it. However, if the removal disconnects the graph, then that node must become part of the solution, and therefore it is also selected. The algorithm finishes when all nodes have been fixed or removed. A pseudo-code description of the algorithm is given in Fig. 11.2.

```

Heuristic for MCDS
1   $D \leftarrow V$  /* initial solution =  $V$  */
2  Set all vertices  $v \in V$  to non-fixed
3  while there is  $v \in D$  s.t.  $v$  is non-fixed do
4       $F \leftarrow \{v \mid v \text{ is not fixed}\}$ 
5       $u \leftarrow v \in F$  s.t.  $\delta(v)$  is minimum
6      if  $\delta(u) > 1$  and  $D \setminus u$  is not connected then
7          Set  $u$  to fixed
8      else
9           $D \leftarrow D \setminus \{u\}$ 
10         for all  $s \in N(u)$  do
11              $\delta(s) = \delta(s) - 1$ 
12         end if there is no  $(u, v) \in E$  s.t.  $v$  is fixed then
13              $w \leftarrow \operatorname{argmax}_{v: (u, v) \in E} \delta(v)$ 
14             Set  $w$  to fixed
15         end
16     end
17 end
18 Return  $D$ 

```

**Fig. 11.2** Heuristic for MCDS

### 11.3.1 Existing Algorithms

Given a graph  $G = (V, E)$ , a dominating set  $D$  is a subset of  $V$  such that any vertex not in  $D$  is adjacent to at least one vertex in  $D$ . Efficient algorithms for computing the MCDS are essential for solving many practical problems, such as finding a minimum size backbone in ad hoc networks. Wireless ad hoc networks appear in a wide variety of applications, including mobile commerce, search and discovery, and military battlefield.

In this chapter, we present an efficient heuristic algorithm for the MCDS problem. The algorithm starts with a feasible solution containing all vertices of the graph. Then it reduces the size of the CDS by excluding some vertices using a greedy criterion. We also discuss a distributed version of this algorithm. The results of numerical testing show that, despite its simplicity, the proposed algorithm is competitive with other existing approaches.

## 11.4 An Algorithm for the MCDS Problem

In this section, we describe an algorithm for the MCDS problem. As we have already mentioned, most existing heuristics for the MCDS problem work by selecting vertices to be a part of the dominating set and adding them to the final solution. We proceed using the inverse method: the algorithm starts with all vertices in the

```

Compute CDS
1   $D \leftarrow V$ 
2   $F \leftarrow \emptyset$ 
3  while  $D \setminus F \neq \emptyset$  do
4     $u \leftarrow \operatorname{argmin}\{\delta(v) \mid v \in D \setminus F\}$ 
5    if  $G[D \setminus \{u\}]$  is not connected then
6       $F \leftarrow F \cup \{u\}$ 
7    else
8       $D \leftarrow D \setminus \{u\}$ 
9      for all  $s \in D \cap N(u)$  do
10        $\delta(s) \leftarrow \delta(s) - 1$ 
11     end if  $N(u) \cap F = \emptyset$  then
12        $w \leftarrow \operatorname{argmax}\{\delta(v) \mid v \in N(u)\}$ 
13        $F \leftarrow F \cup w$ 
14     end
15   end
16 end Return  $D$ 

```

**Fig. 11.3** Compute a CDS

initial CDS. Then, at each step we select a vertex using a greedy method and either remove it from the current set or include it in the final solution. Figure 11.3 (Algorithm) is a formal description of the proposed procedure.

At the initialization stage, we take the set  $V$  of all vertices as the starting CDS (recall that we deal only with connected graphs). In the algorithm, we consider two types of vertices. A fixed vertex is a vertex that cannot be removed from the CDS, since its removal would result in an infeasible solution. Fixing a vertex means that this vertex will be a part of the final dominating set constructed by the algorithm. A nonfixed vertex can be removed only if their removal does not disconnect the subgraph induced by the current solution. At each step of the algorithm, at least one vertex is either fixed or removed from the current feasible solution.

In Fig. 11.3 (Algorithm),  $D$  is the current CDS;  $F$  is the set of fixed vertices. In the beginning,  $D = V$  and  $F = \emptyset$ . At each iteration of the **while** loop of Fig. 11.3 (Algorithm), we select a nonfixed vertex  $u$ , which has the minimum degree in  $G[D]$ . If removing  $u$  makes the graph disconnected, then we clearly need  $u$  in the final solution, and thus  $u$  must be fixed. Otherwise, we remove  $u$  from the current CDS  $D$  and select some neighbor  $v \in N(u)$  to be fixed, in the case that no neighbor of  $u$  has been fixed before. We select a vertex with the highest connectivity to be fixed, since we want to minimize the number of fixed vertices. These steps are repeated while there is a nonfixed vertex in  $D$ . In the following theorem, we show that the algorithm outputs a CDS correctly.

**Theorem 22.** *Figure 11.3 (Algorithm) returns a connected dominating set, and has the time complexity of  $\mathcal{O}(nm)$ .*

*Proof.* We show by induction on the number of iterations that the returned set  $D$  is a connected dominating set. This is certainly true at the beginning, since the graph is connected, and therefore  $D = V$  is a CDS. At each step, we remove the vertex

with minimum degree, only if the removal does not disconnect  $D$ . The algorithm also makes sure that for each removed vertex  $u$  there is a neighbor  $v \in N(u)$  which is fixed. Thus, for each vertex not in  $D$ , there will be at least one adjacent vertex included in the final set  $D$ . This implies that  $D$  is a CDS.

To determine the time complexity of Fig. 11.3 (Algorithm), note that the **while** loop is executed at most  $n - 1$  times, since we either remove or fix at least one vertex at each iteration. At each step, the most expensive operation is to determine if removing a vertex disconnects the graph. To do this, we need  $\mathcal{O}(m + n)$  time, which corresponds to the time needed to run the depth first search (or breadth first search) algorithm. Thus, the total time complexity of Fig. 11.3 (Algorithm) is given by  $\mathcal{O}(nm)$ .

The proposed algorithm can be considered a convenient alternative to existing methods. Some of the advantages can be seen not only in computational complexity but also in other terms. First of all, it is the simplicity of the method. Most algorithms for MCDS start by creating a (not necessarily connected) dominating set, and subsequently they must go through an extra step to ensure that the resulting set is connected. In the case of Fig. 11.3 (Algorithm), no extra step is needed, since connectedness is guaranteed at each iteration. Another favorable consideration is that the algorithm always maintains a feasible solution at any stage of its execution, thus providing a feasible virtual backbone at any time during the computation.

### 11.4.1 A Distributed Implementation

In the implementation of routing algorithms, it is essential to be able to distribute the work among many processors. However, a difficulty with distributed algorithms is the lack of central information about the network. Therefore, it is frequently not possible to make decisions that depend on global information, without exchanging information in some way. A distributed version of the described algorithm has been proposed to address issues raised by the lack of central information. These problems present themselves when the algorithm needs to verify the connectedness of the remaining nodes, after some node is tentatively disconnected.

The distributed version of the algorithm is defined as a set of messages. To flag the start of the algorithm, a message is sent from an initial node. Starting from there, the control flow is spread to neighbor nodes. The distributed algorithm can be described in the following way. Select initially a node with low connectivity (using a version of the leader election algorithm). Label the selected node  $s$ . Then, try to disconnect  $s$  from the remaining network: this is done in response to a TRY-DISCONNECT message. If the node can be safely remote from the network, that is, the network remains connected, then node  $s$  sends the message DISCONNECTED to its neighbors, in response to which other nodes will perform the same steps. Otherwise, the node sends the message NEWDOM to signal it has been set as a dominator, followed by the message TRY-DISCONNECT.

In the distributed version, an additional assumption is adopted. Due to the difficulty of checking connectedness of the resulting graph in a distributed way, it is assumed that each node can be reached in at most  $k$  hops, where  $k$  is a given constant.

The algorithm described has some advantages over two-phase methods. A main advantage is simplicity, since the algorithm is based on one phase only. Second, there is an advantage of connectedness for each intermediate iteration of the algorithm. This ensures that the partial solutions are also valid solutions for the virtual routing problem, and therefore can be used even before the algorithm has finished. In fact, this type of algorithm can stay running constantly in the network, and is therefore a good way of reconfiguring the system after changes caused by movements of the nodes.

## 11.5 Distributed Heuristic

In this section, we discuss a distributed heuristic for the MCDS problem, based on Fig. 11.3 (Algorithm). For ad hoc wireless network applications, algorithms implemented in a noncentralized, distributed environment have great importance, since this is the way that the algorithm must run in practice. Thus, we propose a distributed algorithm that uses a strategy similar to Fig. 11.2 (Algorithm). We describe below how the steps of the algorithm are defined in terms of a sequence of messages.

In the description of the distributed algorithm, we say that a link  $(v, u)$  is an active link for vertex  $v$ , if  $u$  was not previously removed from the CDS. The messages in the algorithm are sent through active links only, since all other links lead to vertices which cannot be a part of the CDS. We assume, as usual, that there is a starting vertex, found by means of some leader election algorithm [125]. It is known that this can be done in  $\mathcal{O}(n \log n)$  time [6]. We also assume that the leader vertex  $v_l$  is a vertex with the smallest number of neighbors. This feature is not difficult to add to the original leader election algorithm, so we will assume that this is the case.

The execution starts from the leader, which runs the `self-removal` procedure. First, we verify if removing this vertex would disconnect the subgraph induced by the resulting set of vertices. If this is the case, we run the `Fix-vertex` procedure, since then the current vertex must be present in the final solution. Otherwise, the `Remove-vertex` procedure is executed.

The `Fix-vertex` procedure will execute the steps required to fix the current vertex in the CDS. Initially, it sends the message `NEWDOM` to announce that it is becoming a dominator. Then, the current vertex looks for other vertices to be considered for removal. This is done based on the degree of each neighbor; therefore, the neighbor vertex with the smallest degree will be chosen first, and will receive a `TRY-DISCONNECT` message.

The `Remove-vertex` procedure is executed only when it is known that the current vertex  $v$  can be removed. The first step is to send the message `DISCONNECTED` to all neighbors, and then select the vertex which will be the dominator

for  $v$ . If there is some dominator in the neighborhood, it is used. Otherwise, a new dominator is chosen to be the vertex with the highest connectivity in  $N(v)$ . Finally, the message SET-DOMINATOR is sent to the chosen vertex.

### 11.5.1 Computational Complexity

Note that in the presented algorithm, the step with highest complexity consists in verifying connectedness for the resulting network. To do this, we run a distributed algorithm which verify if the graph is still connected when the current vertex is removed. An example of such algorithm is the distributed breadth first search (BFS), which is known to run in  $\mathcal{O}(D \log^3 n)$ , where  $D$  is the diameter (length of the maximum shortest path) of the network, and sends at most  $\mathcal{O}(m + n \log^3 n)$  messages [9]. Thus, each step of our distributed algorithm has the same time complexity.

To speed up the process, we can change the requirements of the algorithm by asking the resulting graph to be connected in  $k$  steps for some constant  $k$ , instead of being completely connected. To do this, the algorithm for connectedness can be modified by sending a message with TTL (time to live) equal to a constant  $k$ . This means that after  $k$  retransmissions, if the packet does not reach the destination, then it is simply discarded. The added restriction implies that we require connectedness in at most  $k$  hops for each vertex. We think that this is not a very restrictive constraint, since it is also desirable that paths between vertices are not very long. With this additional requirement, the diameter of the graph can be thought of as a constant, and therefore the resulting time complexity for each step becomes  $\mathcal{O}(\log^3 n)$ . The time complexity of the whole algorithm is  $\mathcal{O}(n \log^3 n) = \tilde{\mathcal{O}}(n)$ . We use the notation  $\tilde{\mathcal{O}}(f(n))$  to represent  $\mathcal{O}(f(n) \log^k n)$ , for some constant  $k$ .

The number of messages sent while processing a vertex is also bounded from above by the number of messages used in the BFS algorithm. Thus, after running this in at most  $n$  vertices, we have an upper bound of  $\mathcal{O}(nm + n^2 \log^3 n)$  (which is  $\tilde{\mathcal{O}}(n^2)$  when the graph is sparse) for the total message complexity. These results are summarized in the following theorem.

**Theorem 23.** *The distributed algorithm with  $k$ -connectedness requirement runs in time  $\mathcal{O}(n \log^3 n) = \tilde{\mathcal{O}}(n)$  and has message complexity equal to  $\mathcal{O}(nm + n^2 \log^3 n)$ . For sparse graphs, the message complexity is  $\tilde{\mathcal{O}}(n^2)$ .*

The details of the resulting algorithm are shown in Fig. 11.4. We prove its correctness in the following theorem.

**Theorem 24.** *The distributed algorithm presented in Fig. 11.4 finds a correct CDS.*

*Proof.* One of the basic differences between the structure of connected dominating sets created by the distributed algorithm and the centralized algorithm is that we now require connectedness in  $k$  steps, that is, the diameter of the subgraph induced by the resulting CDS is at most  $k$ . Of course, this implies that the final solution is connected.

**Fig. 11.4** Actions for a vertex  $v$  in the distributed algorithm

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><u>General actions:</u></p> <p>on TRY-DISCONNECT, do Self-removal</p> <p>on SET-DOMINATOR, do Fix-vertex</p> <p>on NEWDOM, do</p> <p style="padding-left: 20px;">{ dominator <math>\leftarrow</math> source, Fix-vertex }</p> <p><u>Self-removal:</u></p> <p><b>If</b> <math>\delta(v) = 1</math>, <b>then</b></p> <p style="padding-left: 20px;">send message DISCONNECTED to neighbor</p> <p style="padding-left: 20px;">send message SET-DOMINATOR to neighbor</p> <p><b>Else</b>, run distributed BFS algorithm from this vertex.</p> <p style="padding-left: 20px;"><b>If</b> some vertex is not reached <b>then</b></p> <p style="padding-left: 40px;">Fix-vertex</p> <p style="padding-left: 20px;"><b>Else</b></p> <p style="padding-left: 40px;">Remove-vertex</p> <p style="padding-left: 20px;"><b>End-If</b></p> <p><b>End-If</b></p> <p><u>Fix-vertex:</u></p> <p><b>If</b> <math>v</math> is non-fixed, <b>then</b></p> <p style="padding-left: 20px;">set <math>v</math> to fixed</p> <p style="padding-left: 20px;">send message NEWDOM to neighbors</p> <p style="padding-left: 20px;">ask the degree of each non-fixed, non-removed neighbor</p> <p style="padding-left: 20px;">send message TRY-DISCONNECT to neighbors, according to increasing degree order</p> <p><b>End-If</b></p> <p><u>Remove-vertex:</u></p> <p style="padding-left: 20px;">send to active neighbors the message DISCONNECTED</p> <p><b>If</b> there is no dominator, <b>then</b></p> <p style="padding-left: 20px;">ask the degree of active neighbors</p> <p style="padding-left: 20px;">set <math>u</math> to neighbor with highest degree</p> <p><b>Else</b></p> <p style="padding-left: 20px;">set <math>u</math> to dominating neighbor</p> <p><b>End-If</b></p> <p style="padding-left: 20px;">send message SET-DOMINATOR to vertex <math>u</math></p> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

To show that the result is a dominating set, we argue similarly to what was proved for Fig. 11.3 (Algorithm). At each iteration, a vertex will be either removed from the solution or set to be in the final CDS. In the case that a vertex is removed, it must be dominated by a neighbor, otherwise it will send the message SET-DOMINATOR to one of its neighbors. Thus, each vertex not in the solution is directly connected to some other vertex which is in the solution. This shows that the resulting solution is a DS, and therefore a CDS.

Now, we show that the algorithm terminates. First, every vertex in the network is reached because the network is supposed to be connected, and messages are sent from the initial vertex to all other neighbors. After the initial decision (to become

**Table 11.1** Results of computational experiments for instances with 100 vertices, randomly distributed in square planar areas of size  $100 \times 100$  and  $120 \times 120$ ,  $140 \times 140$ , and  $160 \times 160$ . The average solutions are taken over 30 iterations

| Size             | Radius | Average degree | AWF   | BCDP  | Distr. | Nondistr. |
|------------------|--------|----------------|-------|-------|--------|-----------|
| $100 \times 100$ | 20     | 10.22          | 28.21 | 20.11 | 20.68  | 19.18     |
|                  | 25     | 15.52          | 20.00 | 13.80 | 14.30  | 12.67     |
|                  | 30     | 21.21          | 15.07 | 10.07 | 10.17  | 9.00      |
|                  | 35     | 27.50          | 11.67 | 7.73  | 8.17   | 6.30      |
|                  | 40     | 34.28          | 9.27  | 6.47  | 6.53   | 4.93      |
|                  | 45     | 40.70          | 7.60  | 5.80  | 6.13   | 4.17      |
|                  | 50     | 47.72          | 6.53  | 4.40  | 4.77   | 3.70      |
| $120 \times 120$ | 20     | 7.45           | 38.62 | 27.71 | 28.38  | 27.52     |
|                  | 25     | 11.21          | 26.56 | 18.26 | 19.00  | 17.78     |
|                  | 30     | 15.56          | 20.67 | 13.40 | 14.63  | 12.40     |
|                  | 35     | 20.84          | 15.87 | 10.03 | 11.27  | 9.13      |
|                  | 40     | 25.09          | 12.67 | 8.33  | 9.00   | 7.00      |
|                  | 45     | 30.25          | 10.47 | 7.23  | 7.67   | 5.77      |
|                  | 50     | 36.20          | 8.87  | 6.00  | 6.57   | 4.70      |
| $140 \times 140$ | 30     | 11.64          | 25.76 | 18.10 | 18.90  | 17.03     |
|                  | 35     | 15.51          | 20.00 | 13.33 | 14.37  | 12.73     |
|                  | 40     | 19.40          | 16.40 | 10.87 | 11.57  | 9.67      |
|                  | 45     | 23.48          | 13.33 | 9.03  | 9.40   | 7.77      |
|                  | 50     | 28.18          | 11.33 | 7.63  | 8.33   | 6.23      |
|                  | 55     | 33.05          | 9.80  | 6.57  | 7.17   | 5.30      |
|                  | 60     | 38.01          | 8.80  | 5.70  | 6.20   | 4.53      |
| $160 \times 160$ | 30     | 9.14           | 31.88 | 22.20 | 23.20  | 21.88     |
|                  | 35     | 12.21          | 24.50 | 17.07 | 17.36  | 16.18     |
|                  | 40     | 15.63          | 19.73 | 13.47 | 14.07  | 12.43     |
|                  | 45     | 19.05          | 16.33 | 11.07 | 11.40  | 9.93      |
|                  | 50     | 22.45          | 13.80 | 9.47  | 9.67   | 7.93      |
|                  | 55     | 26.72          | 12.20 | 7.80  | 8.33   | 6.87      |
|                  | 60     | 30.38          | 10.33 | 7.10  | 7.47   | 5.80      |

fixed or to be removed from the CDS), a vertex just propagates messages from other vertices, and does not ask further information. Since the number of vertices is finite, this implies that the flow of messages will finish after a finite number of steps. Thus, the algorithm terminates, and returns a correct connected dominating set.

### 11.5.2 Experimental Results

Computational experiments were run to determine the quality of the solutions obtained by the heuristic proposed for the MCDS problem. We implemented both the centralized and distributed versions of the algorithm using the C programming

**Table 11.2** Results of computational experiments for instances with 150 vertices, randomly distributed in square planar areas of size  $120 \times 120$ ,  $140 \times 140$ ,  $160 \times 160$ , and  $180 \times 180$ . The average solutions are taken over 30 iterations

| Size             | Radius | Average degree | AWF   | BCDP  | Distr. | Nondistr. |
|------------------|--------|----------------|-------|-------|--------|-----------|
| $120 \times 120$ | 50     | 54.51          | 9.47  | 6.30  | 6.70   | 4.63      |
|                  | 55     | 63.02          | 7.73  | 5.70  | 6.40   | 4.20      |
|                  | 60     | 71.12          | 6.67  | 4.83  | 5.53   | 4.00      |
|                  | 65     | 81.08          | 6.00  | 3.73  | 4.30   | 3.53      |
|                  | 70     | 89.22          | 5.00  | 3.23  | 3.67   | 3.10      |
|                  | 75     | 98.04          | 4.79  | 3.04  | 2.82   | 2.54      |
|                  | 80     | 104.64         | 4.64  | 2.82  | 2.09   | 2.00      |
| $140 \times 140$ | 50     | 42.91          | 11.60 | 7.60  | 8.33   | 6.13      |
|                  | 55     | 49.75          | 10.07 | 6.87  | 7.40   | 5.20      |
|                  | 60     | 57.74          | 8.33  | 5.87  | 6.80   | 4.47      |
|                  | 65     | 64.70          | 7.60  | 5.27  | 6.20   | 4.17      |
|                  | 70     | 72.04          | 6.93  | 4.83  | 5.50   | 4.00      |
|                  | 75     | 78.82          | 5.87  | 3.87  | 4.53   | 3.60      |
|                  | 80     | 86.55          | 5.47  | 3.33  | 3.93   | 3.33      |
| $160 \times 160$ | 50     | 33.94          | 14.00 | 9.63  | 10.07  | 8.43      |
|                  | 55     | 40.31          | 12.27 | 8.47  | 8.97   | 6.73      |
|                  | 60     | 45.89          | 10.73 | 7.30  | 8.13   | 5.73      |
|                  | 65     | 53.36          | 9.60  | 6.27  | 6.77   | 4.83      |
|                  | 70     | 58.75          | 8.67  | 6.10  | 6.60   | 4.43      |
|                  | 75     | 64.39          | 7.80  | 5.40  | 6.13   | 4.37      |
|                  | 80     | 72.05          | 6.60  | 4.63  | 5.80   | 4.00      |
| $180 \times 180$ | 50     | 27.92          | 17.60 | 11.57 | 12.37  | 10.37     |
|                  | 55     | 33.05          | 15.13 | 10.13 | 10.43  | 8.47      |
|                  | 60     | 38.09          | 12.40 | 8.50  | 9.23   | 7.27      |
|                  | 65     | 43.95          | 11.53 | 7.70  | 8.53   | 6.10      |
|                  | 70     | 48.75          | 10.13 | 7.17  | 7.43   | 5.33      |
|                  | 75     | 55.08          | 9.33  | 6.30  | 6.87   | 4.53      |
|                  | 80     | 60.73          | 8.33  | 5.70  | 6.47   | 4.33      |

language. The computer used was a PC with Intel processor and enough memory to avoid disk swap. The C compiler used was the `gcc` from the GNU project, without any optimization. The machine was running the Linux operating system.

In the computational experiments, the testing instances were created randomly. Each graph has 100 or 150 vertices distributed randomly over an area, varying from  $100 \times 100$  to  $180 \times 180$  square units. The edges of a unit-disk graph are determined by the size of the radius, whose value ranged from 20 to 60 units. The resulting instances were solved by an implementation of Fig. 11.3 (Algorithm), as well as by a distributed implementation, described in the previous section. For the distributed algorithm, we used the additional requirement of  $k$ -connectedness with  $k = 20$ . The algorithms used for comparison are the ones proposed in [6] and [26]. They are referred to in the results (Tables 11.1 and 11.2) as AWF and BCDP, respectively.

The results show that the nondistributed version of Fig. 11.3 (Algorithm) consistently gives results which are not worse than any of the other algorithms. The distributed version of the algorithm gives comparable results, although not as good as the nondistributed version. This can be explained by the fact that the distributed implementation lacks the benefit of global information, used by Fig. 11.3 (Algorithm), for example, always finding the vertex with smallest degree. However, despite the restrictions on the distributed algorithm, it performs very well. It must also be noted that the resulting implementation is very simple compared to the other approaches, and therefore can be executed faster.

## 11.6 Conclusion

In the last few years, the move toward wireless communication devices has accelerated, as popular products have incorporated wireless features, and personal computers and personal digital assistants have become a common place in everyone's lives. With this increase in use, however, comes new challenges for the efficient use of the available broadcast spectrum.

Broadcasting on wireless networks can be modeled using a graph theoretical approach. Nodes in the network are linked to other nodes when a certain distance threshold is reached. This leads to so-called unit-disk graphs.

Once we have a graph representing the wireless network, an interesting problem is to determine the minimum number of nodes that can talk to each other node in the graph. This is known as the MCDS problem.

In this chapter, we describe an approach to the MCDS problem. The algorithm is applied to ad hoc wireless networks, modeled as unit-disk graphs. The algorithm is especially valuable in situations where setup time is costly, since it maintains a feasible solution at any time during the computation and thus can be executed without interrupting the network operation.

A distributed version of the algorithm is also presented, which tries to adapt the basic algorithmic idea to a distributed setting. The experimental results show that both algorithms are able to find high quality solutions with values comparable to those produced by the best existing algorithms.

The above mentioned advantages and the simplicity of the proposed algorithm make it an attractive alternative when solving the MCDS problem in dynamic environments.

# Chapter 12

## Power-Aware Routing in MANETs

### 12.1 Introduction

Operating power is one of the most important resources required by wireless devices [131]. For practical use, wireless devices can only store electricity in relatively small quantities. This makes it necessary to consider conservation measures that reduce consumption of electricity by the equipment.

In a related issue, current technology employed by batteries is not sufficient to power wireless devices for long periods. Thus, energy conservation is one of the few strategies that can really make a difference in the context of mobile device usage.

These properties of wireless systems demonstrate that considering optimal expenditure of energy is an important area of research on wireless systems. Given the scarcity of power as an operational resource on mobile systems, it is important to notice that there is still no satisfactory solution that provides long-term service and/or low power consumption.

#### 12.1.1 Power Control Problem

Considering the limited amount of power available for the operation of wireless units, there is currently high demand for particular techniques that can effectively decrease the amount of power necessary for the operation of wireless devices. Such a solution would ideally allow wireless devices to operate for longer periods and at reduced power consumption levels.

In this chapter, we present an optimization model for the power usage minimization problem, with a formulation that exploits the intrinsic relationship between required power and broadcast reach of wireless units. The resulting model is referred to as the *power control problem in ad hoc networks* (PCADHOC).

To solve this problem effectively, we derive a linear integer programming formulation for the power control problem. The proposed formulation is subsequently

used to compute lower bounds on the amount of required power in the network. The constraints of the problem are demonstrated to guarantee the required resource used by network, such that all provisioned transmissions can be successfully performed.

In the last part of this chapter, a distributed algorithm based on a variable neighborhood search (VNS) strategy is presented. The results of experiments with the algorithm show that the power savings are considerable.

## 12.2 MANETs and Power Management

Mobile ad hoc networks, or MANETs, have been proposed in the last few years as a new technological infrastructure for telecommunication systems. The main goal of such an architecture is to deliver communication services over geographical areas where fixed servers for wireless systems cannot be easily deployed.

The deployment of ad hoc networks on inaccessible areas has made possible a number of applications that depend on the availability of wireless services. Examples of such applications include support for rescue groups, medical applications on remote regions, and real-time support for military operations in battlefield.

On the technological front, the existence of a large variety of new mobile clients, such as cell phones and personal digital assistants (PDAs), has also prompted the rise of interest in MANETs. Some new possibilities involve the creation of personal, peer to peer systems where information can be shared by a small group working on related activities. As a concrete example, consider how a MANET can be used to support groups of scientists sharing information locally during a conference.

### 12.2.1 Resource Limitations

Wireless ad hoc networks have a variety of applications that make them ideal for wide field deployment. Such wireless systems are, however, limited on resources such as battery power. To obtain optimal results of its use, MANETs must employ techniques based on hardware and/or software to minimize resource consumption. One of the objectives of this chapter is to introduce mathematical programming models that can be successfully used to provide improved solutions for power management problems. The mathematical models described here can be applied to compute the necessary amount of power used by the wireless network at a specific moment in time.

The problem described here is called *PCADHOC*. We finally present an algorithmic approach to solve the PCADHOC, in which we assume that a fixed amount of data must be sent from a set of sources to a set of destination nodes. We use the described algorithm to determine the optimal amount of power necessary to deliver data, as requested by the problem.

## 12.3 MANET Architecture

The main principle of MANETs is that each user can act as a client or server for connection purposes. Thus, differently from cellular systems which have dedicated servers responsible for each cell, the MANET architecture depends on the geographical distribution of clients, and its topology cannot be easily predicted.

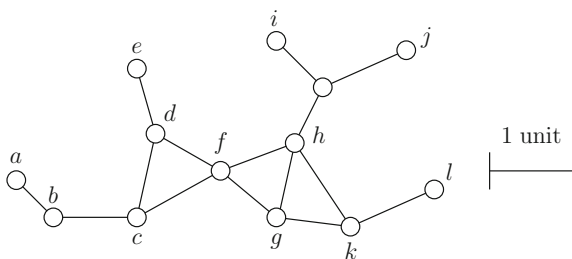
Figure 12.1 gives an example where ad hoc units are represented as nodes in a graph. Each pair of users is connected if they are within the transmission reach of the wireless unit. Assuming that all wireless units have the same capacity, the communication network can be represented as an undirected graph. In a more realistic situation, however, units can have different reach sizes, and the connection links may be asymmetric.

At each time, the topology of a MANET is determined by the position of its components, the wireless units. Since clients can move freely, a number of issues of connectivity among clients must be addressed to maintain communication in the group and guarantee the correct delivery of messages.

One of the main problems in wireless ad hoc networks is related to the determination of power levels on which clients must operate. The following characteristics can be affected by the determination of power levels:

- *Power consumption:* Clearly, to minimize power consumption, the total power level must be reduced on all wireless units. However, if the total power level goes below a threshold, it becomes impossible to route messages, since the network becomes disconnected.
- *Network traffic:* Increasing the power level on units may allow a greater number of units be reached in one transmission operation (hop). This can make possible the delivery of messages with less intermediate steps, which in turn will result in reduced traffic. In a very simple situation, it could be possible to reach every node in the network directly by simply adjusting the power level. This is not possible in practice due to the interference generated.
- *Interference:* At the same time that the neighborhood size increases with power level, the same happens to the interference caused by units using the same frequency. This in practice restricts the amount of power that can be used and the number of neighbors that can be directly reached by a wireless unit.

**Fig. 12.1** Graphical representation of an ad hoc network, where all wireless units have the same transmission reach. Nodes are linked by edges when the distance between them is at most 1



In the area of power control, the main objective is to find methods for minimizing power consumption among wireless units. This must be done in such a way that the minimum amount of energy is used by network clients.

## 12.4 Related Work

We present a quick review of the most interesting results in the area of power control for ad hoc networks. MANETs have been a focus of interest in optimization and distributed computing, due to its topological properties, which result in interesting algorithmic possibilities. For instances, many papers in the area are concerned with the optimal location, and the selection of a subset of nodes to construct a minimally sized connected backbone [26, 28].

Power control techniques have direct impact on routing strategies for MANETS. Therefore, much of the work on power control has been concentrated on the development of new protocols that can minimize the used power. For example, Jung and Vaidya [102] provide a new protocol for power control, based on information available through lower level network layers. Another example of this approach is given in Narayanaswamy et al. [137].

Urpi et al. [184] describe a game theoretic approach for modeling power control in ad hoc networks. The authors analyze mechanisms that can be used to keep cooperation among users, despite the selfish characteristics of such networked organizations. More about game theory applied to MANETs can be seen in MacKenzie and Wicker [124].

A simple strategy for minimizing the power consumption in a network consists of trying to reach the state of minimum power in which the network is still connected. Despite the apparent optimality of this technique, Park and Sivakumar [154] have shown that this is not necessarily an optimum power strategy for power control minimization.

Still another approach for power control is presented by Kawadia and Kumar [103]. Two protocols are proposed, in which the main technique used is clustering of mobile units according to some of its features.

The algorithms mentioned above have in common the fact that they try to handle the problem with relatively low level protocols. The main concern with these techniques is that is difficult to analyze the resulting solution and determine how good it is in comparison with other approaches. The only way to evaluate the quality of solutions is through simulation on a small set of instances. We propose instead a mathematical approach for power control determination, using integer programming [118] and combinatorial techniques.

The advantage of this method is that, defining the problem in a formal way and studying the resulting IP formulations, one can have a better picture of the complexity of its solution. Also, one can put into action a large number of techniques developed to solve this type of problems. Somewhat similar proposals have been

made by Marks II et al. [128] and Das et al. [46]. These, however, consider instead the determination of a tree for data broadcast.

We present in this chapter a distributed algorithm to solve this combinatorial problem. The algorithm is based on the VNS procedure. This has the objective of overcoming the complexity problems introduced by integer formulations, giving a near optimal solution to the problem in reasonable time.

The remaining sections of this chapter are organized as follows. In Sect. 12.5, we present a formulation of PCADHOC. We propose an integer programming formulation, which can be used to derive lower bounds for the problem. In Sect. 12.6, we propose a distributed variable search neighborhood (VNS) algorithm for the PCADHOC. The algorithm will be used to give near optimal solutions for the problem using a distributed strategy. In Sect. 12.7, we give results of computational experiments performed with instances of the problem. Finally, in Sect. 12.8, we give concluding remarks and discuss future work.

## 12.5 Problem Formulation

Initially, we derive a mathematical formulation for the problem, that will be later used to describe an algorithmic approach for power control. We suppose that there is a set of wireless units  $V = \{v_1, \dots, v_n\}$ . For each wireless unit  $v \in V$ , we define as  $p(v) \in \mathbb{R}$  the *power level* at which  $v$  operates. A related concept, the *transmission reach*  $d(v)$  of unit  $v$  is the radius of the area around  $v$  that can be reached by its wireless transmissions. The power level of clients generally determine the transmission reach of the wireless units. In other words, it can be assumed that

$$d(v) = k \cdot p(v).$$

We represent the wireless network by a graph  $G = (V, E)$ , where the set of nodes  $V$  are the network clients. The set of edges  $E(G)$  is in general directed, and depends on the power level of each wireless unit. For example, if unit  $v$  operates at power level  $\alpha$ , then it can reach the set  $N(\alpha, v)$  (also called *transmission neighborhood*) of wireless units around  $v$ . This set has  $\delta(\alpha, v) = |N(\alpha, v)|$  elements.

Clearly, for a fixed graph  $\delta(\alpha, v)$  is dependent on  $d(v)$ . Thus,  $\delta$  is also dependent on the power level, as well as the position of potential neighbors in the network. A simple consequence of the fact that there a finite number of nodes in the network is that we must consider only a finite combination of possible power levels.

**Proposition 2.** *Given a finite set  $V$  of wireless units of the same type, the corresponding set of different transmission neighborhoods is also finite. These transmission neighborhoods are induced by a finite set of power levels  $\mathcal{L}$ . Moreover,  $|\mathcal{L}| \leq \binom{n}{2}$ , for  $n = |V|$ .*

*Proof.* For any fixed node  $v$ , a transmission neighborhood is defined by the power level and the position of the remaining nodes. For a specific configuration of node

positions, let  $x_1, \dots, x_{n-1}$  be the distances between  $v$  and the remaining nodes. The number of different neighborhoods from  $v$  is clearly at most  $n$ . Also, as power level is proportional to transmission neighborhood size, we need just  $n$  possibly different power levels to reach the different neighborhoods at distances  $0, x_1, \dots, x_{n-1}$ .

Now, since each wireless unit  $v$  is of the same type, the power level required to reach node  $w$  from  $v$  is the same to reach node  $v$  from  $w$ . Thus, just  $\binom{n}{2}$  power levels are necessary to describe the system.

Our objective is to find the minimum power level necessary to send  $d$  units of data from source nodes to destination nodes. The resulting problem is combinatorial since, as we have shown, just a finite number of different configurations of power levels exist. Let these levels be called  $l_1, \dots, l_{n_i}$ , for unit  $i$ .

Suppose that there is a finite set of data items  $\gamma_1, \dots, \gamma_k$  that must be sent in the MANET. Each data item has a source node  $s_i$  and a destination  $d_i$  node. Then, we define PCADHOC as the problem of minimizing the sum of the power level in all nodes in the network, subject to the constraints that all data items must be sent. Its is easy to verify that these constraints require that all pairs of source–destination pairs be connected in the resulting network.

A solution for the problem is, therefore, given by a set  $\mathcal{L} = \{l_1, \dots, l_n\}$ , of power levels, one for each node in the network, and, for each data item  $\gamma_1, \dots, \gamma_k$ , a feasible flow of data from  $s_i$  to  $d_i$  in the resulting graph  $G_P$ , induced by the power levels.

### 12.5.1 An Integer Programming Model

Using the assumptions described above, we derive an integer programming model for the PCADHOC. Let the wireless ad hoc network be represented by a graph  $G = (V, E)$ , where nodes are the wireless units. Assume that there is a finite set by  $\mathcal{L} = \{l_1, \dots, l_L\}$  of possible power levels. There is a set of  $k$  messages that must be sent from source  $s_j$  to destination  $d_j$ , for  $j \in \{1, \dots, k\}$ . Then, let  $x_{il}$  be a binary variable defined as

$$x_{il} = \begin{cases} 1 & \text{if node } i \text{ is at the } l\text{th power level} \\ 0 & \text{otherwise.} \end{cases}$$

Also, define the binary variable  $w_e^j$  as

$$w_e^j = \begin{cases} 1 & \text{if edge } e \text{ is part of the } j\text{th path} \\ 0 & \text{otherwise.} \end{cases}$$

Finally, define the binary variable  $y_e$ , for  $e \in V \times V$  as

$$y_e = \begin{cases} 1 & \text{if } e = (i, j) \text{ and node } j \text{ can be reached by } i \\ 0 & \text{otherwise.} \end{cases}$$

If  $p_l$  represents the radius of the transmission reach for power level  $l$ , then a formulation for the PCADHOC can be written as

$$\min \sum_{i=1}^n \sum_{l=1}^L l x_{il},$$

subject to

$$\sum_{l=1}^L x_{il} = 1 \quad \text{for } i \in \{1, \dots, n\} \quad (12.1)$$

$$d(i, j) y_e \leq \sum_{l=1}^L p_l x_{il} \quad \text{for all } (i, j) = e \in E \quad (12.2)$$

$$A w^j \leq b \quad \text{for } j \in \{1, \dots, k\} \quad (12.3)$$

$$w_e^j \leq y_e \quad \text{for all } e \in E, j \in \{1, \dots, k\} \quad (12.4)$$

$$x_{ij} \in \{0, 1\} \quad \text{for } i \in \{1, \dots, n\}, \text{ and } j \in \{1, \dots, k\} \quad (12.5)$$

$$w^i \in \{0, 1\}^E \quad \text{for } i \in \{1, \dots, k\} \quad (12.6)$$

$$y_e \in \{0, 1\} \quad \text{for all } e \in V \times V, \quad (12.7)$$

where  $A = (a)_{ij}$  is a  $n \times n^2$  matrix where all entries are 1 except for the  $a_{ii}$  entries. Also,  $b^i$ , for  $i \in \{1, \dots, k\}$ , is a requirements vector, that is, it has entries  $b_j^i = 1$  for  $j = s_i$ ,  $b_j^i = -1$  for  $j = d_i$  and  $b_j^i = 0$  otherwise.

**Theorem 25.** *The integer program above is a correct formulation for the PCADHOC.*

*Proof.* Starting with the objective function, we know that the reach of transmission is proportional to the amount of power used, thus minimizing the sum of  $p_l x_{il}$  for all  $v_i \in V$  will minimize the total power used.

Constraint (12.1) ensures that for each node exactly one power level is selected. Then, Constraint (12.2) defines the relationship between variables  $x_i$  and  $y_e$ . The variable  $y_e$  will be 1 if and only if a power level with enough capacity is selected for node  $i$  such that  $e = (i, j)$ .

Constraint (12.3) ensures that, for all source–destination pairs, there is a feasible path leading from  $s_i$  to  $d_i$ , for  $i \in \{1, \dots, k\}$ . This guarantees the delivery of the existing data from source  $s_i$  to destination  $d_i$ . We say that these constraints are the *flow constraints*. Constraint (12.4) is then used to ensure that the data flow described in the previous constraint is feasible, with respect to the existing edges. This means that data can be sent through an edge if and only if it really exists in the graph induced by the power levels assigned to each node.

Finally, Constraints (12.5)–(12.7) give domain requirements for the variables used. Thus, a solution to the integer programming formulation above characterizes a feasible solution to the PCADHOC. On the other hand, it is clear that a feasible solution to the PCADHOC will define at least one feasible solution for the IP model above. Therefore, this is a correct formulation for the PCADHOC.

## 12.6 Variable Neighborhood Search Algorithm

The problem described in the previous section is difficult to solve exactly. In practice, however, for many combinatorial problems there exist good algorithms that can give reasonable solutions for most instances. We describe a polynomial algorithm for the PCADHOC as formalized above, and show that it yields good solutions for practical instances.

The technique we propose for solving the power control problem is the VNS algorithm [88, 132]. VNS is a metaheuristic for optimization, and has been applied to NP-hard combinatorial optimization problems such as coloring [8, 11, 199]. VNS is based on the simple idea of employing different neighborhoods in a local search algorithm.

To understand why VNS is effective, one must recall the standard algorithm for local search, shown in Fig. 12.2 (Algorithm). In this heuristic, starting from an initial solution, local changes (also called perturbations) are made until a local optimum is found. However, a solution may be locally optimal with respect to a neighborhood  $V^1$ , but nonoptimal with respect to a second neighborhood  $V^2$ . In fact, any nonglobal solution must be nonoptimal for at least one neighborhood. By using systematically more than one neighborhood in the local search algorithm, it is more difficult to become trapped on a local optimum during the search process.

The VNS algorithm has, therefore, a flow of control similar to a local search algorithm. However, instead of using just one local search neighborhood, a VNS explores a set  $V^1, \dots, V^l$  of neighborhoods of a local minimum solution. The resulting procedure is presented in Fig. 12.3 (Algorithm).

### Local Search

```

1  Initialize solution  $s$ 
2  while the solution can be improved do
3    Do a local perturbation in  $s$ 
4    if  $s$  improved then
5      Accept new solution
6    else
7      Revert  $s$  to old solution
8    end
9  end
```

**Fig. 12.2** Local search algorithm

|                                                                                                                                                                                                                                                                                                    |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Variable Neighborhood Search</b><br>1    Initialize solution $s$<br>2 <b>while</b> termination criterion not met <b>do</b><br>3 <b>for</b> $k \leftarrow 1$ to $l$ <b>do</b><br>4       Perform local search on neighborhood $V^k$ (as shown in Algorithm 12.2)<br>5 <b>end</b><br>6 <b>end</b> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**Fig. 12.3** Variable neighborhood search algorithm

### 12.6.1 Neighborhood Descriptions

Given the VNS scheme described in Fig. 12.3 (Algorithm), we describe the different neighborhoods used in the implementation of the algorithm for the PCADHOC.

- *Neighborhood  $V^1$* . The first local search neighborhood employed, which will be called  $V^1$ , uses the simple technique of, for each node  $v \in V$ , trying to decrease its power level by 1 unit. Then, we select the node  $w \in V$  that gives the biggest improvement. In this definition of neighborhood, a solution is a local optimum if it cannot be improved by simply decreasing the power on one or more wireless units. It is easy to check that this neighborhood must reach an optimum after at most

$$\sum_{i=1}^n l(i),$$

where  $l(i)$  is the power level of node  $v_i \in V$ , since in this case the objective function is monotonically decreasing.

- *Neighborhood  $V^2$* . In this neighborhood, an element  $v \in V$  is randomly selected and its power level is increased to 1 unit. Then, the remaining nodes  $w \in V$ ,  $w \neq v$  are tested, to find the one that can have its power level decreased by at least 1 unit. The algorithm will select the node  $w$  which yields the best value of objective function after decreasing its power level.
- *Neighborhood  $V^3$* . This neighborhood uses the idea of changing the path  $P_i$  linking a source–destination pair of nodes. This is done by selecting a third node in the same connected component, and making  $P_i$  pass through this node. To do this, one can use, for example, the depth first search algorithm. The last step is then to check if it is possible to reduce the power level of some of the affected nodes.
- *Neighborhood  $V^4$* . In this neighborhood, the algorithm selects the node that has the highest rate of power by number of paths served, and try to reduce its power level. Doing so, one has to recompute the paths needed to route the transmitted data.

**Theorem 26.** *The complexity of local search for the neighborhoods described above is, respectively,  $O(n^3)$ ,  $O(n^3)$ ,  $O(n^2)$ , and  $O(kn^2)$ .*

*Proof.* The main operation that must be considered in the algorithms for  $V^1$  and  $V^2$  shown above is that of checking if a solution with different power levels is feasible. Basically, feasibility is equivalent to connectivity among nodes in any source–destination pair  $s_i, d_i$ , for  $i \in \{1, \dots, k\}$ . The feasibility of a solution can be determined by running an algorithm to find paths between each pair of source–destination pairs  $s_i, d_i$ , for  $i \in \{1, \dots, k\}$ . For example, depth first search can be used to find the needed paths. Such algorithms can be executed very efficiently, in time  $O(m + n)$ , where  $m$  is the number of edges in the induced graph. As  $O(m)$  can be at most  $O(n^2)$ , the algorithms for neighborhoods  $V^1$  and  $V^2$  have both complexity  $O(n^3)$ . For neighborhood  $V^3$ , the complexity also depends on an algorithm to find paths, since this operation must be done to change an existing source–destination path. However, this must be run twice, thus the complexity is  $O(n^2)$ . The last neighborhood tries to change the power of one node, and for this it is necessary to check at most  $k$  paths. Thus, the complexity becomes  $O(kn^2)$ .

### 12.6.2 Distributed VNS Algorithm

The method described in the previous section can be used to give good solutions in a sequential and centralized way. However, we can benefit in many ways from a distributed version. The main reason for considering a distributed algorithm is that the power control problem on MANETs is clearly distributed. Thus, each node in the MANET should be able to cooperate to find a solution which can determine its power level.

A way of distributing the computational effort of the algorithm described in the last section is to have each node computing the solution for part of the network. We propose an algorithm where a smaller part of the graph representing the network is considered initially. This information is then used to compute power levels for the part of the graph that was considered.

This can be done in the following way. Initially, each node can send messages to its neighbors up to a distance  $K$ , using the facilities of the routing protocol for the ad hoc network. Then, the neighbors will return information corresponding to its position. This data can subsequently be used by the node to compute a set of power levels based on a local computation using the sequential algorithm.

A problem that must be solved, however, is that not all information may be available locally when the decisions are made by the distributed algorithm. In particular, the information about sources and destinations, required to ensure the constraints of the problem, may not be available. On the other hand, in the sequential algorithm, there is global information available about all nodes in the network.

To compensate for this lack of global information, we can have some hints based on the MANET routing protocol. For example, it can be required for each source or destination node  $w$ , whose position is currently unknown to the local node, that an acknowledgment message be sent. Then, the last node in the route from  $w$  to the current node can be added as a source (or destination). This information can, therefore, be used to approximate the data for the global network.

**Distributed Algorithm**

```

1  Construct a graph  $G$  representing the neighborhood of node  $v$  in the network
2  for each source or destination  $w$  not in  $G$  do
3      Send a message to  $w$  and receive its acknowledgment
4      Let  $u \in G$  be the node through which the acknowledgment was received
5      Make  $u$  to become a source or destination, depending on the type of node  $w$ 
6  end
7  Run the VNS algorithm on the graph  $G$  computed above
8  for each neighbor do
9      Ask the resulting value of its solution
10 end if neighbors have power levels less than the value calculated locally then
11     Let  $P_l$  be the sum of the power levels calculated by neighbors
12     Let  $P_n$  be the sum of the power levels calculated locally.
13     Increase the local power level by  $P_n/P_l$ 
14 end

```

**Fig. 12.4** Distributed algorithm for the PCADHOC

A last step in the distributed algorithm consists of keeping the consistency of the final solution. This is done by checking the solution found by neighbor nodes. If the power level for each neighbor is at least equal to the value computed locally, then no changes are necessary. However, if the value is less than what was computed locally, the power level of the current node can be increased by an amount proportional to  $P_n/P_l$ , where  $P_l$  is the sum of power levels calculated by neighbor nodes and  $P_n$  is the sum of power levels calculated locally for these neighbors. The formal version of the distributed algorithm as discussed above is shown in Fig. 12.4 (Algorithm).

## 12.7 Computational Experiments

The algorithms presented in the previous sections were implemented and tested to determine the quality of resulting solutions. Instances were created using a generator similar to the one used in [26]. The generator receives as input the number of wireless units and the area covered by these units. The instance is then composed of uniformly distributed points over the region.

### 12.7.1 Experimental Settings

The algorithm proposed was implemented using the C language and compiled using the GCC compiler. The optimization level used in the GCC compiler was `-O2`. The integer programming formulation was implemented using the Xpress Mosel [98] software. The implemented programs were run in a PC with 256 MB of memory and a 2.7 GHz processor.

**Table 12.1** Results of computational experiments performed with the PCADHOC instances

| Instance |     | Relaxed IP |          | VNS   |      | Distrib. VNS |      |
|----------|-----|------------|----------|-------|------|--------------|------|
| $n$      | $k$ | LB         | Time (s) | Value | Time | Value        | Time |
| 40       | 4   | 27.4       | 17.6     | 29.3  | 8.3  | 30.1         | 5.4  |
| 50       | 6   | 49.1       | 28.2     | 53.8  | 13.0 | 53.8         | 7.0  |
| 60       | 8   | 73.8       | 466.9    | 77.6  | 15.5 | 78.1         | 8.7  |
| 80       | 10  | 94.4       | 86.6     | 98.2  | 18.1 | 97.5         | 12.1 |
| 90       | 12  | 183.2      | 127.5    | 192.0 | 25.3 | 130.7        | 15.1 |
| 100      | 14  | 272.7      | 259.1    | 285.1 | 41.2 | 286.0        | 19.3 |
| 120      | 16  | 317.0      | 361.6    | 326.5 | 27.4 | 327.2        | 22.7 |
| 140      | 18  | 395.6      | 408.0    | 403.8 | 35.4 | 405.2        | 21.4 |

### 12.7.2 Lower Bound on Power Consumption

The integer programming model introduced in the previous section can become very difficult to solve exactly using existing techniques such as enumerative methods. However, they can be used to derive a lower bound for the problem, that can be later compared to feasible solutions to the problem. With this information, one can determine a lower bound, useful on determining how far is the given solution from the optimal solution. The integer programming model shown above is used with this objective, and the results will be described below.

### 12.7.3 Integer Programming Results

The IP formulation was run for a number of instances. However, just for very small instances we could find optimal results. This reflects the complexity of the problem, since a large number of variables must be defined to find a feasible solution. The most interesting use of the IP formulation was as a lower bounding procedure. With this information, we could determine for some problems how far from optimal the solution given by the heuristic is.

### 12.7.4 Heuristic Results

A summary of results using the VNS procedure is presented in Table 12.1. The columns in this table describe, respectively, the problem instance, the lower bound found by the relaxation of the IP formulation, the results given by the VNS algorithm, and its execution time, the result of the distributed algorithm and the execution time. Results for the heuristics were averaged for a number of 20 runs per

instance. A similar method was used for execution times. However, in this case we used the time until the best solution was found (the total running time, on the other hand, was the same for all runs, equal to 10 min).

From the results stated in Table 12.1, one can see that the VNS algorithm was able to find good solutions for the majority of the instances. The performance on each instance was very good for instances with size ranging from 40 to 140 nodes. The quality of the solutions given by the distributed algorithm has also shown to be similar to the sequential version. Moreover, it proved to give better results in terms of computational time. This is due to the reduced size of the problems that must be solved in the distributed algorithm.

## 12.8 Conclusion

In this chapter, we considered the problem of power control on wireless ad hoc networks. This problem is very important in the definition of optimal strategies for data routing in a distributed environment where many users with limited power must have access to the network. To determine power levels of nodes connected to the ad hoc network, a combinatorial approach was considered. The main restriction introduced concerns the ability of sending a required piece of data from selected sources to destinations.

We discussed an integer programming model representing the problem. We also considered a variable neighborhood strategy that gives near optimal results for a set of test instances.

Many research questions remain in this area. For example, it would be interesting to define an extended model where interference is taken into consideration. Also, to complement the computational aspect of the study started in this paper, it could be possible to define other distributed algorithms with smaller time complexity and/or message complexity. Another interesting issue is to study the dynamics of power control, such as changes in position, inclusion and exclusion of new wireless units, etc.



# Appendix A

## Concepts in Communication Networks

In the study of algorithms for telecommunications problems, several basic concepts related to the operation of communications networks have to be taken into consideration. Although this book does not require a deep understanding of network systems, we need to discuss at least the basic concepts in the area of computer networks. This will help us to fully understand the requirements of the algorithms that will be later presented.

This goal of this chapter, therefore, is to provide a basic introduction to the concepts that will be required the remaining parts of the book. Notice that we do not intend by any means to present a comprehensive list of network concepts in general. We try to simplify concepts whenever possible, and avoid the discussion of hardware and platform-related information that would distract from the main goal of this chapter.

### A.1 Computer Networks

A *computer network* is a set of computational devices connected using a signaling medium, which can be either a set of cables, radio waves, or optical media. The devices must understand at least one protocol of interaction, so that information sent on the network is understood by all its participants. The computational devices are denoted as the *nodes* of the network, and connection media is referred to as *network links*.

As explained in this definition, computer networks are systems formed by the interaction of several physical and logical parts. The different ways in which the physical layers (the hardware) can be organized are referred to as the possible *physical topologies* of the network.

### A.1.1 Network Topology

Common network topologies include the case in which all nodes are directly connected – this is also known as a (*full*) *mesh*. A full mesh is not scalable, since it is extremely hard to create all the necessary connections when the number of nodes increase. Realistic networks have connections among a selected subset of pairs of nodes.

A *bus topology* is the one in which all nodes are connected to a common bus. In this case, all nodes share the communication medium, and must be careful to access it only when the message can be successfully sent. The bus topology is commonly used given its simplicity. The use of a *medium access protocol* (MAC) is frequently necessary to allow the communication among nodes attached to the bus.

Another common topology for computer networks is a *ring*. In this topology, nodes connect to each other in the following way. Given a predetermined sequence of the nodes, the first node in the sequence is connected to the second. The  $i$ th node is connected to the  $(i + 1)$ th node until the last node is connected to the first, forming a cycle. In this topology, since the transmission medium is not completely shared, nodes are required to agree with who has the right to send messages. A common way to determine this is using a *token*, which determines what node has the right to send messages. Ring topologies where a token is used to control access to the medium are called *token ring* networks [90].

Another simple network topology is the *star* topology, where all nodes connect to a central node, which is responsible to retransmit messages. In this case, the central node needs to have enough capacity to support all request on behalf of the nodes. Although it is a simple topology, it is not adequate in general because any failure in the central node can stop the entire network. Moreover, it is not scalable since the central node has limited capacity.

To avoid the problems of the simpler network topologies described above, modern networks have general topologies that are formed by a mix of the described topologies as well as more unconventional ones. All these complex structures must be kept together by efficient and robust protocols and algorithms. For example, the Internet is nowadays a complex arrangement of networks ordered in the most diverse ways and using different media, ranging from copper cables to optical fiber and radio waves.

Concerning our view of network topology, the topology can be easily represented as a graph, that is, a set of nodes and edges (see Appendix A), where edges represent the existing physical connection between nodes. Such graphs are used in most network algorithm for find information about the network such as routes, costs, and reliability, for example.

### A.1.2 Data Packets and Circuits

There are two fundamental ways of exchanging information in a computer network, which are represented by the concepts of *data packets* and *virtual circuits*.

A *data packet* is a unit of information that is sent through a network. When data is transmitted using data packets, there is no requirement saying that all packets must follow the same path in the network. While this is possible, it may be that different packets that compose the data are sent through different paths in the network. In a *packet switching* network, all information sent is subdivided into smaller packets, each with roughly the same size. This allows data to be sent without producing too much latency on the underlying medium. It also allows for the use of different paths for redundancy and better use of the available network bandwidth.

The use of data packets configures what is called the *packet switching* model of network communication. Packet switching is now the dominating model of network communication, since most protocols, such as the Internet protocol, use packet switching to implement data transfer.

An alternative way of sending information in a network is known as *circuit switching*. A virtual circuit is a precomputed path in a network connecting a source to a destination. Circuit switching works by creating a fixed path (a virtual circuit) that can be used to route all information that needs to be sent between a pair of nodes. Although easy to understand, circuit switching is not a very common way of sending information on modern networks because it can waste resources.

For example, while a virtual circuit is established, there may be other nodes that want to access the same network path. This is difficult to arrange with circuit switching. More importantly, with circuit switching, all information must follow the same virtual circuit. This ignores the possibility that other paths may be available to transmit part of the data.

Because of the issues described above, most modern networks are based on packet switching. For this reason, the algorithms described in this book assume generally that information is sent from source to destination(s) in small packets.

### A.1.3 Protocols

A protocol is a set of rules that must be followed by partners during a communication process. Without a protocol, messages sent on a network have no meaning, therefore, connection between nodes cannot be established and as consequence, no information is transferred. Protocols are a required part of the logical structure of a computer network, as discussed in the first section.

Several protocols have been proposed as the main logical connection in computer networks. Examples include the transmission control protocol (TCP) and Internet protocol (IP), the OSI layers of protocols, DECnet, and AppleTalk. While these protocols differ in the way connections and messages are handled, they all allow the logical connection of nodes participating in a network.

The most well-known set of protocols nowadays is the *Internet protocol suite*, mostly known as TCP/IP. The IP was created as the practical set of rules for connection on the Internet project, which later became the Internet we know today.

The set of protocols of the Internet is divided into several subprotocols, the most important of which are the TCP and IP, which give name the IP suite.

The TCP handles the logical connection of nodes. It is responsible for making sure that a logical connection is established, maintained throughout the communication, and properly terminated. It also allows for verification and correction of errors in the transmission. The IP provides the routing part of the IP, that is, determining the exact paths that must be followed by packets in the network. Thus, IP is the most interesting part of the IP for our purposes.

### ***A.1.4 Protocol Stacks and Layers***

Given the similar purpose and design of many network protocols, a logic way of organizing and teaching them has been proposed by the International Standards Organization in the late 1970s. This logical organization is in the form of a stack of protocols, collectively known as open systems interconnection (OSI).

The OSI architecture is composed of seven layers, which determine all possible areas that can be covered by a protocol, in order of abstraction. Although the seven layers can be seen as a simplification (or in some cases a complication) of the real set of protocols in any platform, they represent a common classification view that has been used with great success in the last decades.

The seven layers of the OSI architecture are as follows:

1. *Physical layer*: This is the most lower level layer, where the protocol is used to interpret and generate signals on the hardware level. This protocol layer involves communication using media such as cables (optical or metallic) and radio waves, network cards, and software directly attached to the device, such as the software drivers for network cards.
2. *Data link layer*: This is the second layer of control, where the objective is to treat information as data, not as physical signal. Therefore, this level is responsible for the integrity of the data at its lowest level, such as guaranteeing that the data sent by a computer will be received by the ones that are directly connected to it. Thus, basic error correction is performed at this level to allow communication among neighboring devices.
3. *Network layer*: This layer is the first one where data is treated as a global entity, instead of a piece of data sent in the local network. Thus, the network layer needs to assign an address structure to network, and implement methods for transferring data addressed to a particular node. This process is called *routing*, and is the main task that is described in this book. The network layer must also make sure that the transfer of data is done with the quality required by other layers in terms of flow and reliability.
4. *Transport layer*: The transport layer is responsible for implementing a transparent connection between nodes in the network, using the network layer as its basis. The transport protocol must be able to establish a connection, define the partners

in the connection, and terminate the connection at the request of the interested nodes. The transport layer must also guarantee the quality of the connection, for example, making sure that no packets are lost during the transfer.

5. *Session layer.* The session layer has the responsibility of creating the illusion of a session being established between the partners in the communication. This involves saving the state of the connection so that it can be recovered in case of failures.
6. *Presentation layer.* This layer is used to translate the diverse presentation media available for applications. For example, it needs to translate different character encodings such as ASCII and EBCDIC into a single form that is understandable by the host computer.
7. *Application layer.* The application layer is where the network application is implemented. This includes the vast majority of end user protocols such as http, email, ftp, bittorrent, etc.

### A.1.5 Routing

As we have seen, routing is a part of the third layer of the OSI protocol, the network layer. It is also a crucial part of the whole stack of protocols, since it allows connectivity between nodes that are not in a local network. The Internet as we know would not be possible without an addressing and routing scheme for connectivity between nodes in all parts of the world.

From the mathematical point of view, routing can be seen as an algorithmic process that manipulates the network of users and links, such that packets sent by one of the nodes can follow a path to the destination using only allowed links. To study this process, we use graph theoretical techniques, such as path following and tree construction algorithms, to name a few.

Given a network with underlying graph  $G$ , and a set of nodes  $A \subseteq V$ , a route among nodes in  $A$  is a subset  $E'$  of  $E$  such that all nodes in  $A$  are connected in  $G[E']$ . One of the simplest types of route occurs when  $|A| = 2$ . In this case, the route is defined by a path between nodes  $u$  and  $v$ , for  $u, v \in A$ .

Routing methods can be classified into the following three categories:

- *Unicast routing.* This is the case where there is only one source and one destination for each piece of data routed in the network. This is the most common case of communication for which network protocols have been specified. The result of a unicast routing protocol is a path connecting source and destination in a way that satisfies the requirements of both nodes in terms of speed and reliability. Given the nature of the decisions involved in unicast routing, there are nowadays several good methods and efficient algorithms to create routes satisfying these requirements – the success of the Internet has demonstrated the quality of these routing strategies.

- *Broadcast routing.* In broadcast mode, a node sends information that is supposed to be received by all nodes connected to it. This mode of operation is common in local networks, especially the ones where the nodes are attached to a bus, such as in the bus topology (an example is provided by Ethernet networks).

On the other hand, broadcast in upper levels of the network (such as in a wide-area network) is very wasteful of resources and is normally not implemented. Broadcast is more important for local communication and for wireless systems, where this is the main way of reaching other nodes to which they can be connected.

- *Multicast routing.* Multicast routing is a compromise between unicast and broadcast routing. In this method of operation, the goal is to reach more than one, and possibly a large number of nodes in a network. However, in multicast routing we guarantee that all nodes interested in the data will receive the information, within the limits of delay and reliability imposed by the system. To be successful, multicast protocols should avoid that data be sent to all nodes, thus an addressing and routing scheme must be created to reach only the interested partners within the list of multicast nodes.

Routing protocols also have variations according to the underlying media used by the network. The first main type of such variants are *protocols for wired networks*, using optic, coaxial, and other types of cable. These protocols encompass most of the traditional technology used to connect nodes to the Internet.

The second type of variant of routing protocols is used with *wireless ad hoc networks*. In this case, routing is more difficult due to the lack of a fixed structure, which requires constant measures to ensure connectedness and proper delivery of packets.

### A.1.6 *Quality of Service*

Routing on networks can be analyzed according to different performance parameters, which vary according to the application and aspect of the system that is being optimized.

One of the basic concepts using in networks is that of *quality of service* (QoS). QoS is a guarantee that the requested service will be provided within certain parameters, at least for the majority of the time. QoS can have different meanings for different users, therefore, it be measured according to a variety of parameters.

Some of most common measures for QoS are as follows:

- *Percentage of time the route is active.* This is often called reliability, since it measures how reliable the route is. Reliability can usually be determined as the product of the reliability of the links that are part of a path.
- *Delay.* One of the main features required by users of a multimedia conference system is a low delay in image transmission. Several other applications have similar demands, therefore, the total delay experienced by users should be

reduced. To traverse a link, each packet of data incurs a specific *delay* due to technological constraints. The total *delay*  $\text{delay}(\mathcal{P})$  incurred by a path  $\mathcal{P}$  is equal to the summation of all paths associated with the edges in  $\mathcal{P}$ . Thus,

$$\text{delay}(\mathcal{P}) = \sum_{e \in \mathcal{P}} \text{delay}(e).$$

- *Additional bandwidth available.* Some applications have a variable quantity of data that must be transmitted, according to dynamic changes on the demands of the service. To accommodate such applications, additional bandwidth may be required in each path used to connect nodes.
- *Throughput.* The *throughput* of a network link is defined as the maximum amount of data that can be normally transferred through this link. The throughput of a path is the minimum of the throughput over the links that are part of the path. A QoS requirement may request that the throughput of a route be more than some threshold.
- *Hop distance.* The *network distance* between two nodes, also referred to as the hop distance, is the number of nodes separating the source from the destination. For two paths with similar features, the one with lower hop distance is usually less susceptible to failures, thus it is usually preferred. Hop distance can be used as a measure of QoS when this is the intended goal or the routing process.

### A.1.7 Algorithmic Performance

The practice of routing on networks depends on the existence of efficient algorithms for computing “good” routes. Such routes must satisfy the connectivity criterion, that is, they connect the source node to its destination. The routes must also satisfy any additional QoS requirement.

Algorithms to find such routes can be classified according to algorithmic performance measures. These are formal methods to define how efficient an algorithm may be in terms of its use of resources. The main performance measures used in algorithm analysis are as follows:

- *Time complexity.* This is a measure of the time an algorithm takes, as a function of input size.
- *Message complexity.* Message complexity is number of messages required to communicate with other nodes to perform a distributed algorithm.
- *Space complexity.* Memory usage is the total memory required by the algorithm to perform its required computations.

In addition, algorithms for creating routes can also be analyzed according to different parameters, such as scalability.

Finding the best route according to all performance parameters is a difficult problem from the computational complexity point of view. Therefore, most networks

solve just part of the problem, by finding only a reasonable solution. For example, to find minimum distance estimates for route computation, routers commonly have option that allows the assignment of “weights” to network interfaces. These weights are used in practice to compute routing costs, since real costs are usually too hard to compute.

## A.2 Main Algorithmic Problems

In the context of communications networks, algorithms play an important role in implementing the infrastructure of protocols. Fast protocols require the development of efficient algorithms that can handle the large amount of traffic on modern networks.

To overcome such efficiency constraints, classes of algorithms have been developed to serve the needs of network practitioners. Optimization algorithms have an important role in this scenario. In this section, we present the most well-known classes of optimization problems occurring in communication networks, and how they are approached in terms of research and development.

### A.2.1 Network Design

The first phase of the development of networks is planning: designing the number of nodes, their locations, and the general topology [123]. Designing a network requires numerous considerations, including:

- *Amount of bandwidth.* Each network is designed for a specific set of applications, which require a given network bandwidth to work properly. Although the bandwidth available is directly related to hardware issues, it is also a key requirement for the development of software. The algorithms implemented for the network must have enough efficiency to handle the load over each node.
- *General topology.* The network topology influence several aspects of network design. Servers must be logically and physically located in the regions where their services will most required. Enough bandwidth must be available between clients and servers, and the time necessary to contact them must be within the limits imposed by the application. All these parameters are affected by the network topology chosen during the design phase.
- *Reliability.* Network applications differ a great deal in their reliability requirements. Some applications are less sensitive to problems, specially if connectivity can be recovered quickly. On the other hand, several applications require a reliable connection during their whole period of operation. To ensure such requirements, the network design must provide for reliability measures, such as reserve of bandwidth, back up servers, and the use of cache mechanisms [73].

- *Responsiveness.* Modern network applications require not only correctness and reliability but also timeliness in the results to provide a good user experience. Responsiveness is strictly required in real time applications, but also desirable in most productivity and business network products. Ensuring responsiveness is a key parameter in network design, and must be provided with a combination of topology, hardware, and software measures.

In algorithmic optimization, network design has always been an important area of research. Network design problems have been at the heart of fundamental algorithmic problems, such as computing spanning trees, Steiner trees, shortest paths, and other combinatorial structures. Many of the basic problems in graph theory arise from the need of network designs with a certain structural property.

In modern optimization literature, network design is a constant source of new problems, since it must be solved for each new technology and topology of network. Different of network have requirements that are conflicting, therefore, algorithms must be developed for each version of network in which we are interested.

Network design problems have some features that set them apart, when compared to other types of optimization problems on networks. A special feature of such problems is the availability of more time for their solution, allowing for better decision making than in standard on line problems. In general, when a network is designed, the goal is to make it as efficient as possible and enough time is allowed to find the best possible design. Therefore, it is not uncommon to see algorithms for network design that are allowed several hours or even days to compute the best possible solution.

The better time availability allows for the use of a larger class of algorithm than would be possible otherwise. In particular, partial enumeration algorithms such as Branch-and-bound and Branch-and-cut [139] are commonly employed. Such algorithms not only provide solutions for network design problems, but also try to prove that the solution is optimal. They can also give extra information about the gap between the optimal solution and the best solution found by the algorithm.

In this book, we explore some network design problems with application on multicast routing such as the network packing problem [34], the classical Steiner tree problem [194], the related point-to-point connection problem [43]. The cache placement problem is a type of network design problem where the goal is to minimize the number of caches [144]. It is another design problem that will be later discussed in detail in this book.

### A.2.2 *Packet Routing*

While network design problems are important during the phase of initial requirements of a network system, during its operation, it is fundamental to have highly efficient algorithms able to perform the required network operations in real time.

Among the several tasks in a network system, routing is the most fundamental from the logical point of view, since it allows the connectivity of nodes in the system. Therefore, it must be performed in the best way possible to conserve the resources of the network such as bandwidth.

Packet routing is the second major problem that will be studied in this book. We will initially provide a theoretical description of the major algorithms available for routing between two fixed nodes in what is called *unicast routing*. Algorithms for unicast routing include well-known methods such as Dijkstra's algorithm and Bellman–Ford's algorithm.

The major routing problem described in this book consists of computing routes on a multicast network. Since this problem is known to be NP-hard, we study algorithm that provide near optimal solutions, involving partial enumeration, approximation algorithms, and heuristics.

### A.2.3 *Algorithmic Performance and Complexity*

When considering different algorithms for the same task, it is natural to ask what algorithm is the most efficient according to some criteria. Common parameters for deciding efficiency of an algorithm consist of time, memory, number of messages to other processors, as well as related computational resources.

Despite the simplicity of this question, comparing algorithms is not always an easy task. One can directly measure the resources discussed about for specific runs of an algorithm, but nothing guarantees that such data can be extrapolated to larger instances. In fact, even for instances of comparable size, we may have different performance measures for the same algorithm.

This leads to the need of a more general method for comparing the usage of resources in different algorithms. Such measure is commonly referred to in the computer science literature as computational complexity of an algorithm. The computational complexity can be related to any one of the several computational resources available: time, memory, network messages, etc.

The *input size* for a given problem is a natural measure of the size of the problem that must be solved by an algorithm. In a graph problem, for example, this usually corresponds to the number of nodes and edges. In a list processing problem, this would correspond to the size of the list. In number theory problems, this corresponds to the integer number considered.

The *time complexity* of an algorithm is measured as the number of steps of an algorithm, and this is usually given as a function of the input size. Several special cases of interest exist. An algorithm is said to run in *polynomial time* if the number of steps necessary to run the algorithm can be expressed as a polynomial on the input size.

Complexity measures are usually represented using the big-Oh notation, where only the main exponent of a function is considered. Therefore, some of the main complexity classes are denoted as  $O(\log n)$ ,  $O(n)$ ,  $O(n \log n)$ ,  $O(n^2)$ ,  $O(n^3)$ , and  $O(2^n)$ . As a notational convention, constant complexity values are denoted by  $O(1)$ .

A notable class of problems is the so-called *NP-hard* class. For such problems, only algorithms with time complexity at least  $O(2^n)$  are known. It is also strongly conjectured that no better algorithm exists at all. Moreover, it can be proved that if a polynomial time algorithm exists to any member of *NP-complete*, a large subclass of NP-hard, then all NP-complete problems can be solved in polynomial time [76].

Throughout this book, we are concerned with finding algorithms for some NP-hard problems occurring in multicast routing. Given the complexity features of such problems, described above, we often are forced to provide nonoptimal algorithms to achieve results in reasonable time. Therefore, we are willing to accept nonoptimal results in exchange for efficient running time. For each algorithm described, we will provide to determine the complexity of the algorithm and if the algorithm is exact or not, whenever this information is available.

For more information on complexity issues in algorithms, one can consult standard text books in algorithms such as [42, 108, 146].

## A.2.4 Heuristic Methods

Heuristic algorithms provide a way to avoid the high complexity cost of exact algorithms for NP-Hard problems. Since such problems can be solved exactly in polynomial time any if  $P = NP$ , we know that it may be more appropriate to consider heuristics that work well for common instances of the problem.

Traditionally, heuristics have been proposed for many of the NP-Hard problems, with varying degrees of success.

While most heuristics are specifically tailored for a problem, a class of heuristics that can be applied to several problems is called a metaheuristic. Metaheuristics provide mechanisms of intensification and diversification that can be used to improve the solution generated.

### A.2.4.1 Heuristics

As explained above, some problems are known to be difficult to solve in terms of time complexity, the most notorious being the class of NP-hard problems. To solve such problems, a special type of algorithms called *heuristics* have been used. A heuristic is an algorithm that provides a feasible solution without a guarantee of optimality. Despite this, solutions generated by heuristics are usually very good, and in many cases are close to the optimal for a given problem.

Although particular heuristics have been successful in practice, the process by which they are able to find near optimal solutions is not clear, and their

approximation properties are difficult to determine analytically. In this case, experimental results are found using a set of instances known to be difficult for conventional algorithms. After the set of problem instances is prepared, the results returned by the heuristic are compared either to a known optimal solution or to other heuristics. This type of comparison leads to the discover of heuristics that perform well for a large set of target instances.

In this book, we present several examples of heuristics for routing problems in multicast networks. In fact, heuristics are the major type of algorithm available for such problems, primarily due to the fact that many routing problems are NP-hard in general. Therefore, most of the researchers dealing with this issue have to resort to computational experimentation to assess the quality of the proposed heuristics. We try to present a comparison of such results whenever direct comparison between algorithms is possible.

#### A.2.4.2 Metaheuristics

Since heuristic methods have been very successful in solving practical instances of NP-hard problems, they have raised the quest of general principles in heuristic design that can be applied to a large number of problems in multiple application areas.

As a result of this type of research, a new generation of heuristics have been created based on general properties shared by a large number of heuristics. Such principles, applicable to several NP-hard and combinatorial problems have become known as metaheuristics.

A metaheuristic is a sequence of general steps that when applied to a problem generate a particular instance of a heuristic. Some of the most well-known metaheuristics are *genetic algorithms* [1, 2], *tabu search* [77, 78], *simulated annealing* [31, 106], and *greedy randomized adaptive search procedure* (GRASP) [66, 68].

Some of the advantages of metaheuristics include the simplicity most schemes used in metaheuristic design, and the reuse of methods implemented for other problems in the context of a new problem.

Most heuristics are based on the computational simulation of natural processes. For example, genetic algorithms [2] try to simulate the natural selection mechanism when solving a combinatorial optimization problem. Simulated annealing [31], on the other hand, simulates the process of metal cooling, which is called annealing. Through this ordered process, the goal is to minimize an objective function represented as the energy level of the particles in the annealing process. Each step of the algorithm reduces the amount of energy, until a solution is found at the end of the cooling schedule.

Other metaheuristics are based on generalized search methods. Tabu search, for example, is an extension of a simple “down hill” algorithm to find the optimum of an optimization function [77]. Tabu search introduces nonimproving moves whenever there is no way to strictly improve the current solution. Moreover, it adds what is called tabu moves, that is, solution that should not be repeated during the search phase to avoid cycling.

The GRASP metaheuristic [66] extends local search in another way, by providing a “smart” constructor for initial solutions. The metaheuristic repeats a loop where solutions are constantly generated using a greedy method, and then improved until a local optimum is found. GRASP is one of the most successful metaheuristics, having been applied to several problems in areas including network routing and design. In this book, we present examples of GRASP algorithms being applied to problems in multicast routing.

### A.2.5 *Distributed Algorithms*

A distributed algorithm is a set of steps required to solve a problem, where the steps must be cooperatively performed by more than one processor. In a distributed algorithm, all nodes in the network cooperate for the final result of the algorithm and information is exchanged among nodes using a message passing protocol. The algorithm must have a termination criterion and the final solution must be available for a nonempty subset of the nodes in the network.

Distributed algorithms are essential for the solution of network related problems, since the problems are by nature distributed among several processors. For example, routing is a task that is better handled by a distributed algorithm, since multiple nodes are involved in the process.

Distributed algorithms are frequently used to describe network protocols. Although it is possible to have a central authority that determines what will occur in the network, this is not a scalable solution and requires that all nodes are willing to submit to the decisions of some central authority. Such a model is clearly not the one adopted by the Internet or any other major network.

A distributed algorithm is described as a set of messages and internal computations in response to predefined messages. Therefore, the messages exchanged between processors define the protocol of interaction. The complexity of a distributed algorithm is a function not only of the time and memory required to perform it, but also by the number of messages exchanges needed. The number of messages exchanges required by an algorithm is usually called its *message complexity*.

When working with distributed computation, some basic algorithms are repeatedly used. Some examples are: algorithm for leader election (finding a node in the network with a specific property), algorithm for spanning tree construction (a graph connecting all nodes), depth first search, and breadth first search. All these problems have efficient distributed implementations. In this book, we will explore several distributed algorithms for problems in routing in multicast networks.

## A.3 Additional Resources

Computer networking is a very dynamic field of study. There is a large number of books in this subject that can give a much better description of the basic concepts.

The classic reference in computer networks is Tanenbaum [182]. Other books of note include Stallings [177], Halsall [86], and Bertsekas and Gallager [23].

Most of the research in computer network has appeared in conferences and specialized journals of the area. Among many well-known journals in the area of algorithms for telecommunications networks, we can cite the following:

- *International Journal of Computer and Telecommunications Networking*
- *Networks*
- *IEEE Journal on Selected Areas in Communications*
- *Computer Communication Review*
- *Computer Communications*
- *IEEE-ACM Transactions on Networking*
- *Journal of Combinatorial Optimization*

Due to the dynamic nature of networking, many of the results in routing and networking in general appear in annual conferences. Some of the conferences that are specialized or have specialized sections in networking problems are the following:

- *IEEE Conference on Computer Communications (INFOCOM)*
- *ACM Special Interest Group on Data Communications Conference (SIGCOMM)*
- *ACM International Conference on Mobile Computing and Networking (MOBI-COM)*
- *IEEE Global Communications Conference (GLOBECOM)*
- *IEEE Annual Symposium on Foundations of Comp. Science (FOCS)*

# References

1. E. Aarts and J. Korst. *Simulated annealing and Boltzmann machines: a stochastic approach to combinatorial optimization and neural computing*. Wiley, Chichester, 1990
2. E. H. L. Aarts. Simulated annealing. In *Simulated Annealing and Boltzmann machines: a stochastic approach to combinatorial optimization and neural computing*. Wiley, Chichester, 1989
3. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1993
4. R. Aiex, S. Binato, and M. Resende. Parallel GRASP with path-relinking for job shop scheduling. *Parallel Computing*, 29:393–430, 2003
5. A. B. A.J. Frank, L.D. Wittie. Multicast communication on network computers. *IEEE Software*, 2(3):49–61, 1985
6. K. M. Alzoubi, P.-J. Wan, and O. Frieder. Distributed heuristics for connected dominating set in wireless ad hoc networks. *IEEE ComSoc/KICS Journal on Communication Networks*, 4(1): 22–29, 2002
7. S. Arora and C. Lund. Hardness of approximations. In D. Hochbaum, editor, *Approximation Algorithms for NP-hard Problems*. PWS Publishing, Boston, MA, 1996
8. C. Avanthay, A. Hertz, and N. Zufferey. A variable neighborhood search for graph coloring. *European Journal of Operational Research*, 151:379–388, 2003
9. B. Awerbuch and D. Peleg. Network synchronization with polylogarithmic overhead. In *Proc. 31st Symp. Found. Computer Science*, pages 514–522, New York, NY, 1990. IEEE Computer Society
10. B. S. Baker. Approximation algorithms for NP-complete problems on planar graphs. *Journal of the ACM (JACM)*, 41(1):153–180, 1994. ISSN 0004-5411. doi:<http://doi.acm.org/10.1145/174644.174650>
11. B. Balasundaram and S. Butenko. Graph domination, coloring and cliques in telecommunications. In *Handbook of Optimization in Telecommunications*, pages 865–890. Kluwer, Dordrecht, 2006
12. M. Baldi, Y. Ofek, and B. Yener. Adaptive real-time group multicast. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, page 683, San Francisco, CA, 1997. IEEE
13. A. Ballardie. Core based trees (CBT) multicast routing architecture, RFC 2201 – IETF network working group. On-line document: <http://www.ietf.org/>, 1997
14. A. Ballardie, P. Francis, and J. Crowcroft. Core-based trees (CBT) – an architecture for scalable inter-domain multicast routing. *Computer Communication Review*, 23(4):85–95, 1993
15. Z. Baoxian, L. Yue, and C. Changjia. An efficient delay-constrained multicast routing algorithm. In *International Conference on Communication Technologies (ICCT 2000)*, page S07.2, San Francisco, CA, 2000. IEEE

16. V. C. Barbosa. *An Introduction to Distributed Algorithms*. MIT, 1996
17. F. Bauer. *Multicast routing in point-to-point networks under constraints*. PhD thesis, University of California, Santa Cruz, 1996
18. F. Bauer and A. Varma. Degree-constrained multicasting in point-to-point networks. In *Proceedings IEEE INFOCOM '95, The Conference on Computer Communications*, pages 369–376, San Francisco, CA, 1995. IEEE
19. F. Bauer and A. Varma. ARIES: A rearrangeable inexpensive edge-based on-line Steiner algorithm. *IEEE Journal of Selected Areas in Communications*, 15(3):382–397, 1997
20. M. Bazaraa, J. Jarvis, and H. Sherali. *Linear Programming and Network Flows*. Wiley, New York, NY, 2nd edition, 1990
21. R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957
22. R. E. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958
23. D. Bertsekas and R. Gallager. *Data networks (2nd ed.)*. Prentice-Hall, Upper Saddle River, NJ, USA, 1992
24. K. Bharath-Kumar and J. Jaffe. Routing to multiple destinations in computer networks. *IEEE Transactions on Communications*, 31(3):343–351, 1983
25. D. Blokh and G. Gutin. An approximate algorithm for combinatorial optimization problems with two parameters. *Australasian J. Combin.*, 14:157–164, 1996
26. S. Butenko, X. Cheng, D.-Z. Du, and P. M. Pardalos. On the construction of virtual backbone for ad hoc wireless network. In S. Butenko, R. Murphey, and P. M. Pardalos, editors, *Cooperative Control: Models, Applications and Algorithms*, pages 43–54. Kluwer, Dordrecht, 2002
27. S. Butenko, X. Cheng, C. A. Oliveira, and P. M. Pardalos. A new algorithm for connected dominating sets on ad hoc networks. In S. Butenko, R. Murphey, and P. Pardalos, editors, *Recent Developments in Cooperative Control and Optimization*, pages 61–73. Kluwer, Dordrecht, 2003
28. S. Butenko, X. Cheng, C. A. S. Oliveira, and P. M. Pardalos. Recent developments in cooperative control and optimization. In S. Butenko, R. Murphey, and P. M. Pardalos, editors, *Cooperative Control: Models, Applications and Algorithms*, pages 61–73. Kluwer Academic, Netherlands, 2003
29. K. L. Calvert, E. W. Zegura, and M. J. Donahoo. Core selection methods for multicast routing. In *IEEE ICCCN '95*, pages 638–642, Las Vegas, Nevada, Sept. 1995. IEEE
30. V. Cavalcante and P. Souza. Solving the job shop schedule problem by asynchronous teams. In *Proc. of Int. Symposium on Operations Research with Applications in Engineering, Technology and Management (ISORA'95)*, 1995
31. V. Cerny. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *J. Opt. Theory Appl.*, 45(1):41–51, 1985
32. G. Chen, M. Houle, and M. Kuo. The Steiner problem in distributed computing systems. *Information Sciences*, 74(1):73–96, 1993
33. S. Chen and N. S. S.N. Talukdar. Job-shop-schedule by an A-Team of asynchronous agents. In *IJCAI-93 Workshop on Knowledge-Based Production, Scheduling and Control*, Chambéry, France, 1993
34. S. Chen, O. Günlük, and B. Yener. Optimal packing of group multicasting. In *Proc. IEEE INFOCOM '98*, pages 980–987, San Francisco, CA, 1998. IEEE
35. C. Cheng, R. Riley, S. Kumar, and J. Garcia-Luna-Aceves. A loop-free extended bellman-ford routing protocol without bouncing effect. *ACM Computer Commun. Rev.*, 19(4):224–236, 1989
36. X. Cheng, X. Huang, D. Li, W. Wu, and D.-Z. Du. A polynomial-time approximation scheme for minimum connected dominating set in ad hoc wireless networks. *Networks*, 42(4):202–208, 2003
37. G. V. Chockler, N. Huleihel, I. Keidar, and D. Dolev. Multimedia multicast transport service for groupware. In *TINA Conference on the Convergence of Telecommunications and Distributed Computing Technologies*, pages 43–54, San Francisco, CA, 1996. IEEE

38. C. Chow. On multicast path finding algorithms. In *Proc. IEEE INFOCOMM '91*, pages 1274–1283, San Francisco, CA, 1991. IEEE
39. S.-J. Chung, S.-P. Hong, and H.-S. Huh. A fast multicast routing algorithm for delay-sensitive applications. In *IEEE GLOBECOM'97*, pages 1898–1902, San Francisco, CA, 1997. IEEE
40. W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and A. Schrijver. *Combinatorial Optimization*. Wiley-Interscience, New York, 1997
41. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT, MA, 1990
42. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT, MA, 1992
43. R. Correa, F. Gomes, C. A. Oliveira, and P. M. Pardalos. A parallel implementation of an asynchronous team to the point-to-point connection problem. *Parallel Computing*, 29(4): 447–466, 2003
44. R. C. Correa. A synchronizer for partially synchronous parallel computations. Technical report, Universidade Federal do Ceará, Brazil, 2001
45. Y. Dalal and R. Metcalfe. Reverse path forwarding of broadcast packets. *Communications of the ACM*, 21(12), 1978
46. A. Das, R. Marks II, M. El-Sharkawi, P. Arabshahi, and A. Gray. Minimum power broadcast trees for wireless networks: integer programming formulations. In *Proc. IEEE Infocom 2003, San Francisco, CA*, San Francisco, CA, 2003. IEEE
47. B. Das and V. Bharghavan. Routing in ad-hoc networks using minimum connected dominating sets. In *International Conference on Communications*, pages 376–380, San Francisco, CA, 1997. IEEE
48. S. Deering. Multicast routing in internetworks and extended LANs. In *ACM SIGCOMM Summer 1988*, pages 55–64, Vancouver, BC, Canada, 1988. Association for Computing Machinery
49. S. Deering. *Multicast Routing in a Datagram Network*. PhD thesis, Stanford University, 1991
50. S. Deering and D. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Transactions on Computer Systems*, pages 85–111, 1990
51. S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C.-G. Liu, and L. Wei. An architecture for wide-area multicast routing. *Computer Communication Review*, 24(4):126–135, 1994
52. S. Deering, D. L. Estrin, D. Farinacci, V. Jacobson, C.-G. Liu, and L. Wei. The PIM architecture for wide-area multicast routing. *IEEE/ACM Transactions on Networking*, 4(2): 153–162, 1996
53. S. Deering, D. Estrin, D. Farinacci, H. Handley, A. Helmy, V. Jacobson, C. Liu, P. Sharma, and L. Wei. Protocol independent multicast-sparse mode (pim-sm): protocol specification, 1998. RFC (experimental) 2362, Internet Engineering Task Force
54. R. Diestel. *Graph Theory*. Springer, third edition, 2005
55. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1:269–271, 1959
56. C. Diot, W. Dabbous, and J. Crowcroft. Multipoint communication: a survey of protocols, functions, and mechanisms. *IEEE Journal on Selected Areas in Communications*, 15(3), 1997
57. M. Doar and I. Leslie. How bad is naive multicast routing. In *Proceedings of the IEEE INFOCOM*, pages 82–89, San Francisco, CA, 1993. IEEE
58. D. Du and P. Pardalos, editors. *Network Optimization Problems: Algorithms, Complexity and Applications*. World Scientific, Singapore, 1993
59. D.-Z. Du and P. M. Pardalos. Subset interconnection designs: Generalizations of spanning trees and Steiner trees. In *Network Optimization Problems*, pages 111–124. World Scientific, Singapore, 1993
60. D.-Z. Du, B. Lu, H. Ngo, and P. M. Pardalos. Steiner tree problems. In C. Floudas and P. Pardalos, editors, *Encyclopedia of Optimization*, volume 5, pages 227–290. Kluwer, Netherlands, 2001
61. C. Ellis, S. Gibbs, and G. Rein. Groupware: Some issues and experiences. *Commun. ACM*, 34(1):39–58, 1991
62. M. Eltoweissy, M. Heydari, L. Morales, and I. Sudborough. Combinatorial optimization of group key management. *Journal of Network and Systems Management*, 12(1):33–50, 2004

63. H. Eriksson. MBONE: The multicast backbone. *Communications of ACM*, 37(8), 1994
64. U. Feige. A threshold of  $\ln n$  for approximating set cover. *Journal of the ACM*, 45(4):634–652, 1998
65. G. Feng and T. P. Yum. Efficient multicast routing with delay constraints. *International Journal of Communication Systems*, 12:181–195, 1999
66. T. Feo and M. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8:67–71, 1989
67. T. Feo and M. Resende. Greedy randomized adaptive search procedures. *J. of Global Optimization*, 6:109–133, 1995
68. T. Feo, M. Resende, and S. Smith. A greedy randomized adaptive search procedure for maximum independent set. *Operations Research*, 42:860–878, 1994. URL <http://www.research.att.com/~mgcr/doc/gmis.ps.Z>
69. D. Ferrari and D. C. Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, 8:368–379, 1990
70. P. Festa. Shortest path algorithms. In *Handbook of Optimization in Telecommunications*, pages 185–210. Kluwer, Dordrecht, 2006
71. L. Ford. Network flow theory. Paper p-923, RAND Corporation, Santa Monica, CA, 1956
72. A. Forsgren and M. Prytz. Dimensioning multicast-enabled communications networks. *Networks*, 39:216–231, 2002
73. A. Forsgren and M. Prytz. Telecommunications network design. In *Handbook of Optimization in Telecommunications*, pages 269–290. Kluwer, Dordrecht, 2006
74. H. Fujinoki and K. J. Christensen. The new shortest best path tree (sbpt) algorithm for dynamic multicast trees. In *24th Conference on Local Computer Networks*, pages 204–211, San Francisco, CA, 1999. IEEE
75. R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Programming Languages and Systems*, 5(1):66–77, 1983
76. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco CA, 1979
77. F. Glover. Future paths for integer programming and links to artificial intelligence. *Comp. and Operations Res.*, 5:533–549, 1986
78. F. Glover and M. Laguna. *Tabu Search*. Kluwer, July 1997
79. M. X. Goemans and D. P. Williamson. A General Approximation Technique for Constrained Forest Problems. *SIAM J. Comp.*, 24, 1995
80. F. Gomes, C. Menezes, A. Lima, and C. Oliveira. Asynchronous organizations for solving the point-to-point connection problem. In *Proc. of the Intl. Conference on Multiagents Systems (ICMAS)*, pages 144–149, Paris, France, 1998. IEEE Computer Society
81. F. C. Gomes, A. G. Lima, C. A. S. Oliveira, and C. N. Meneses. Asynchronous organizations for solving the point-to-point connection problem. In *Proceedings of International Conference on Multi-Agent Systems (ICMAS)*, Paris, France, July 1998. IEEE
82. F. C. Gomes, P. M. Pardalos, C. A. Oliveira, and M. G. Resende. Reactive GRASP with path relinking for channel assignment in mobile phone networks. In *Proceedings of the 5th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, pages 60–67, Rome, Italy, 2001. ACM
83. S. R. Gorti, S. Humair, R. D. Sriram, S. Talukdar, and S. Murthy. Solving constraint satisfaction problems using a-teams. *AI-EDAM*, 10:1–19, 1996
84. S. Guha and S. Khuller. Approximation algorithms for connected dominating sets. *Algorithmica*, 20(4):374–387, 1998
85. S. Gupta and P. Srimani. Adaptive core selection and migration method for multicast routing in mobile ad hoc networks. *IEEE transactions on parallel and distributed systems*, 14(1), 2003
86. F. Halsall. *Data communications, computer networks and open systems (4th ed.)*. Addison Wesley, Redwood City, CA, USA, 1995
87. L. Han and N. Shahmehri. Secure multicast software delivery. In *IEEE 9th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE'00)*, pages 207–212, San Francisco, CA, 2000. IEEE

88. P. Hansen and N. Mladenovic. An introduction to VNS. In S. Voss, S. Martello, I. Osman, and C. Roucairol, editors, *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*. Kluwer, Dordrecht, 1998
89. A. Helmy and S. Gupta. Fotg: fault-oriented stress testing of ip multicast. *IEEE Communications Letters*, 9(4):375–7, 2005
90. M. Henningsson, K. Holmberg, and D. Yuan. Ring network design. In *Handbook of Optimization in Telecommunications*, pages 291–311. Kluwer, Dordrecht, 2006
91. S. Hong, H. Lee, and B. H. Park. An efficient multicast routing algorithm for delay-sensitive applications with dynamic membership. In *Proceedings of IEEE INFOCOM'98*, pages 1433–1440, San Francisco, CA, 1998. IEEE
92. H. B. Hunt III, M. Marathe, V. Radhakrishnan, S. Ravi, D. Rosenkrantz, and R. Stearns. NC-approximation schemes for NP- and PSPACE-hard problems for geometric graphs. *J. Algorithms*, 26:238–274, 1998
93. F. Hwang. A unifying approach to determine the necessary and sufficient conditions for nonblocking multicast 3-stage clost networks. *IEEE Transactions on Communications*, 53(9): 1581–6, 2005
94. F. Hwang and D. Richards. Steiner tree problems. *Networks*, 22:55–89, 1992
95. F. Hwang, D. Richards, and P. Winter. *The Steiner tree problem*, volume 53 of *Annals of Discrete Mathematics*. North-Holland, Amsterdam, 1992
96. Y. Im, Y. Lee, and Y. Choi. A delay constrained distributed multicast routing algorithm. *Computer Communications*, 20(1), 1997
97. M. Imase and B. Waxman. Dynamic Steiner tree problems. *SIAM J. Discrete Math.*, 4: 369–384, 1991
98. D. O. Inc. *Xpress-Optimizer Reference Manual*. Englewood Cliffs, NJ, 2003
99. X. Jia. A distributed algorithm of delay-bounded multicast routing for multimedia applications in wide area networks. *IEEE/ACM Transactions on Networking*, 6(6):828–837, 1998
100. X. Jiang. Routing broadband multicast streams. *Computer Communications*, 15(1):45–51, 1992
101. R. Judson. Genetic algorithms. In C. Floudas and P. Pardalos, editors, *Encyclopedia of Optimization*, pages 218–218. Springer, 2nd edition, 2009
102. E. Jung and N. H. Vaidya. A power control mac protocol for ad hoc networks. In *ACM MOBICOM 2002*, Atlanta, USA, 2002. ACM
103. V. Kawadia and P. Kumar. Power control and clustering in ad hoc networks. In *IEEE INFOCOM'03*, San Francisco, CA, 2003. IEEE
104. J. L. Kennington and R. V. Helgason. Minimum cost network flow algorithms. In *Handbook of Optimization in Telecommunications*, pages 147–162. Kluwer, Dordrecht, 2006
105. C. Kheong, D. Siew, and G. Feng. Efficient setup for multicast connections using tree-caching. In *Proceedings IEEE INFOCOM 2001, The Conference on Computer Communications*, pages 249–258, San Francisco, CA, 2001. IEEE
106. S. Kirkpatrick, C. D. G. Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983
107. J. Klinecicz. Optimization issues in quality of service. In *Handbook of Optimization in Telecommunications*, pages 435–458. Kluwer, Dordrecht, 2006
108. D. E. Knuth. *The Art of Computer Science: Fundamental Algorithms*. Addison-Wesley, 3rd edition, 1997
109. V. Kompella, J. Pasquale, and G. Polyzos. Multicasting for multimedia applications. In *Proceedings of IEEE INFOCOM'92*, pages 2078–2085, San Francisco, CA, 1992. IEEE
110. V. Kompella, J. Pasquale, and G. Polyzos. Two distributed algorithms for the constrained Steiner tree problem. In *Proceedings of the Second International Conference on Computer Communications and Networking (ICCCN'93)*, pages 343–349, San Francisco, CA, 1993. IEEE
111. V. Kompella, J. Pasquale, and G. Polyzos. Optimal multicast routing with quality of service constraints. *Journal of Network and Systems Management*, 4(2):107–131, 1996

112. V. P. Kompella, J. C. Pasquale, and G. C. Polyzos. Multicast routing for multimedia communication. *IEEE/ACM Trans. Networking*, 1(3):286–292, 1993
113. A. Konak and A. Smith. Network reliability optimization. In *Handbook of Optimization in Telecommunications*, pages 735–760. Kluwer, Dordrecht, 2006
114. L. Kou, G. Markowsky, and L. Berman. A fast algorithm for Steiner trees. *Acta Informatica*, 15:141–145, 1981
115. J. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(48):50, 1956
116. G. Kumar, N. Narang, and C. Ravikumar. Efficient algorithms for delay bounded multicast tree generation for multimedia applications. In *6th International Conference on High Performance Computing*, pages 169–173, Berlin, 1999. Springer
117. J. Kurose and A. K. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison Wesley, 1999
118. E. Lee and D. Lewis. Integer programming for telecommunications. In *Handbook of Optimization in Telecommunications*, pages 67–102. Kluwer, Dordrecht, 2006
119. B. Levine and J. Garcia-Luna-Aceves. A comparison of reliable multicast protocols. *Multimedia Systems*, 6(5):334–348, 1998
120. B. Li, M. Golin, G. Italiano, X. Deng, and K. Sohraby. On the optimal placement of web proxies in the internet. In *Proceedings of IEEE INFOCOM 99*, San Francisco, CA, 1999. IEEE
121. C. Li, S. McCormick, and D. Simchi-Levi. The point-to-point delivery and connection problems: complexity and algorithms. *Discrete Applied Math.*, 36:267–292, 1992
122. H. Longo. Solving the set covering and partition problems using A-Teams. Master’s thesis, UNICAMP (State University of Campinas, Brazil), 1995 (In portuguese)
123. H. Luna. Network planning. In *Handbook of Optimization in Telecommunications*, pages 213–240. Kluwer, Dordrecht, 2006
124. A. MacKenzie and S. Wicker. Game theory and the design of self-configuring, adaptive wireless networks. *IEEE Communications Magazine*, 39(11):126–131, 2001
125. N. Malpani, J. Welch, and N. Vaidya. Leader election algorithms for mobile ad hoc networks. In *Proc. Fourth International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, pages 96–103, Boston, MA, 2000. ACM
126. Z. Mao, D. Johnson, O. Spatscheck, J. E. van de Merwe, and J. Wang. Efficient and robust streaming provisioning in VPNs. In *Proceedings of the WWW2003 (May 20-24 2003), Budapest, Hungary*, Budapest, Hungary, 2003. WWW03
127. M. V. Marathe, H. Breu, H. B. Hunt III, S. S. Ravi, and D. J. Rosenkrantz. Simple heuristics for unit disk graphs. *Networks*, 25:59–68, 1995
128. R. J. Marks II, A. K. Das, M. ElSharkawi, P. Arabshahi, and A. Gray. Minimum power broadcast trees for wireless networks: Optimizing using the viability lemma. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, San Francisco, CA, 2002. IEEE
129. S. Martins and C. Ribeiro. Metaheuristics and applications to optimization problems in telecommunications. In *Handbook of Optimization in Telecommunications*, pages 103–128. Kluwer, Dordrecht, 2006
130. C. Meneses. Private communication, 2001. Research on UNICAMP (State University of Campinas, Brazil)
131. M. Min and A. Chinchuluun. Optimization in wireless networks. In *Handbook of Optimization in Telecommunications*, pages 891–915. Kluwer, Dordrecht, 2006
132. N. Mladenović and P. Hansen. Variable neighborhood search. *Computers and Operations Research*, 24:1097–1100, 1997
133. M. F. Mokbel, W. A. El-Haweet, and M. N. El-Derini. A delay constrained shortest path algorithm for multicast routing in multimedia applications. In *Proceedings of IEEE Middle East Workshop on Networking*, San Francisco, CA, 1999. IEEE
134. J. Moy. OSPF version 2, RFC 1583 – IETF network working group. On-line document: <http://www.ietf.org/>, last accessed on June 1, 2004, 1994

135. J. Moy. Multicast extensions to OSPF, RFC 1584 – IETF network working group. On-line document: <http://www.ietf.org/>, last accessed on June 1, 2004, 1994
136. J. T. Moy. *OSPF – Anatomy of an Internet routing protocol*. Addison Wesley, Reading, MA, 1998
137. S. Narayanaswamy, V. Kawadia, R. Sreenivas, and P. Kumar. Power control in ad-hoc networks: Theory, architecture, algorithm and implementation of the compow protocol. In *European Wireless Conference – Next Generation Wireless Networks: Technologies, Protocols, Services and Applications*, pages 156–162, Florence, Italy, 2002. EW2005
138. M. Natu and S. C. Fang. Network Loading and Connection Problems. Technical Report 308, North Carolina State University, 1995
139. G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. Wiley, New York, 1988
140. C. Noronha and F. Tobagi. Optimum routing of multicast streams. In *IEEE INFOCOM 94*, pages 865–873, San Francisco, CA, 1994. IEEE
141. Y. Ofek and B. Yener. Reliable concurrent multicast from bursty sources. *IEEE Journal on Selected Areas in Communications*, 15(3):434–444, 1997
142. C. A. Oliveira and P. M. Pardalos. A distributed optimization algorithm for power control in wireless ad hoc networks. In *Proc. 18th Int. Parallel and Distributed Processing Symposium (IPDPS'04), volume 7: 6th Workshop on Advances in Parallel and Distributed Computational Models (WAPDCM'04)*, page 177. IEEE Computer Society, 2004. April 26th 2004, Santa Fe, New Mexico
143. C. A. Oliveira, P. M. Pardalos, and M. Resende. GRASP with path-relinking for the QAP. In *5th Metaheuristics International Conference*, pages 57.1–57.6, Kyoto, Japan, 2003. MIC03
144. C. A. Oliveira, O. A. P. Panos M. Pardalos, and M. G. Resende. Streaming cache placement problems: Complexity and algorithms. *Int. J. of Computational Science and Engineering*, 3(3):173–183, 2007
145. C. Papadimitriou and M. Yannakakis. Optimization, approximation, and complexity classes. *J. Comput. System Sci.*, 43:425–440, 1991
146. C. H. Papadimitriou. *Computational complexity*. Addison Wesley, Reading, MA, 1994
147. C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization*. Prentice Hall, Englewood Cliffs, NJ, 1982
148. P. Pardalos and B. Khoury. An exact branch and bound algorithm for the steiner problem in graphs. In D.-Z. D. M. Li, editor, *Proceedings of COCOON'95 (Xi'an, China, August 24-26, 1995)*, volume 959 of *Lecture Notes in Computer Science*, pages 582–590. Springer, Berlin, 1995
149. P. Pardalos and B. Khoury. A heuristic for the steiner problem on graphs. *Comp. Opt. & Appl.*, 6:5–14, 1996
150. P. Pardalos, B. Khoury, and D.-Z. Du. A test problem generator for the steiner problem in graphs. *ACM Transactions on Mathematical Software*, 19(4):509–522, 1993
151. P. Pardalos, F. Hsu, and S. Rajasekaran, editors. *Mobile Networks and Computing*, volume 52 of *DIMACS Series*. American Mathematical Society, Providence, RI, 2000
152. P. M. Pardalos and D. Du, editors. *Network Design: Connectivity and Facilities Location*, volume 40 of *DIMACS Series*. American Mathematical Society, Providence, RI, 1998
153. J. Park and C. Park. Development of a multi-user & multimedia game engine based on TCP/IP. In *Proceedings of IEEE Pacific Rim Conference on Communications Computers and Signal Processing*, pages 101–104, Victoria, BC, Canada, 1997. IEEE
154. S.-J. Park and R. Sivakumar. Load sensitive transmission power control in wireless ad-hoc networks. In *IEEE Global Communications Conference (GLOBECOM'02)*, Taipei, Taiwan, 2002. IEEE
155. J. Pasquale, G. Polyzos, and G. Xylomenos. The multimedia multicasting problem. *ACM Multimedia Systems Journal*, 6(1):43–59, 1998
156. P. Paul and S. V. Raghavan. Survey of multicast routing algorithms and protocols. In *Proceedings of the Fifteenth International Conference on Computer Communication (ICCC 2002)*, San Francisco, CA, 2002. IEEE

157. R. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957
158. V. Priwan, H. Aida, and T. Saito. The multicast tree based routing for the complete broadcast multipoint-to-multipoint communications. *IEICE Transactions on Communications*, E78-B(5), 1995
159. M. Prytz. *On optimization in Design of Telecommunications Networks with Multicast and Unicast Traffic*. PhD thesis, Dept. of Mathematics, Royal Institute of Technology, Stockholm, Sweden, 2002
160. M. Prytz and A. Forsgren. Dimensioning of a multicast network that uses shortest path routing distribution trees. Paper trita-mat-2002-os1, Department of Mathematics, Royal Institute of Technology, Stockholm, Sweden, 2002
161. S. Ramanathan. Multicast tree generation in networks with asymmetric links. *IEEE/ACM Trans. Networking*, 4(4):558–568, 1996
162. M. G. C. Resende and C. C. Ribeiro. A GRASP with path-relinking for private virtual circuit routing. *Networks*, 41:104–114, 2003
163. J. Rexford. Route optimization in ip networks. In *Handbook of Optimization in Telecommunications*, pages 679–700. Kluwer, Dordrecht, 2006
164. G. N. Rouskas and I. Baldine. Multicast routing with end-to-end delay and delay variation constraints. In *IEEE INFOCOM'96*, pages 353–360, San Francisco, CA, 1996. IEEE
165. S. Sabri and B. Prasada. Video conferencing systems. *Proc. of the IEEE*, 73(4):671–688, 1985
166. H. F. Salama, D. S. Reeves, and Y. Viniotis. Shared multicast trees and the center selection problem: A survey. Tr-96/27, Dept. of Electrical and Computer Engineering, NCSU, 1996
167. H. F. Salama, D. S. Reeves, and Y. Viniotis. A distributed algorithm for delay-constrained unicast routing. In *Proc. IEEE INFOCOM'97*, Kobe, Japan, 1997. IEEE
168. H. F. Salama, D. S. Reeves, and Y. Viniotis. The delay-constrained minimum spanning tree problem. In *2nd IEEE Symposium on Computers and Communications (ISCC '97)*, pages 699–704, New York, NY, 1997. IEEE Computer Society
169. H. F. Salama, D. S. Reeves, and Y. Viniotis. Evaluation of multicast routing algorithms for real-time communication on high-speed networks. *IEEE Journal on Selected Areas In Communications*, 15(3), 1997
170. L. A. Sanchis. Experimental analysis of heuristic algorithms for the dominating set problem. *Algorithmica*, 33:3–18, 2002
171. C. Semeria and T. Maufer. Introduction to IP multicast routing. Internet draft (IETF), 1996
172. A. Shaikh and K. G. Shin. Destination-driven routing for low-cost multicast. *IEEE Journal of Selected Areas in Communications*, 15(3):373–381, 1997
173. J. M. Simon Doar. *Multicast in the Asynchronous Transfer Mode Environment*. PhD thesis, St. John's College, University of Cambridge, 1993
174. M. S. Sloman and X. Andriopoulos. Routing algorithm for interconnected local area networks. *Computer Networks and ISDN Systems*, 9(2):109–130, Feb. 1985
175. R. Sriram, G. Manimaran, and C. Siva Ram Murthy. Algorithms for delay-constrained low-cost multicast tree construction. *Computer Communications*, 21(18):1693–1706, 1998
176. R. Sriram, G. Manimaran, and C. Siva Ram Murthy. A rearrangeable algorithm for the construction of delay-constrained dynamic multicast trees. *IEEE/ACM Transactions on Networking*, 7(4):514–529, 1999
177. W. Stallings. *Data and computer communications*. Macmillan, Indianapolis, IN, USA, 1985
178. I. Stojmenovic, M. Seddigh, and J. Zunic. Dominating sets and neighbor elimination based broadcasting algorithms in wireless networks. In *Proc. IEEE Hawaii Int. Conf. on System Sciences*, San Francisco, CA, 2001. IEEE
179. Q. Sun and H. Langendoerfer. Efficient multicast routing for delay-sensitive applications. In *Proceedings of the Second Workshop on Protocols for Multimedia Systems (PROMS'95)*, pages 452–458, Salzburg, Austria, 1995. PROMS
180. V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: practice and experience*, 2(4):315–339, Dec. 1990
181. H. Takahashi and A. Matsuyama. An approximate solution for the Steiner problem in graphs. *Mathematica Japonica*, pages 573–577, 1980

182. A. Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 2002
183. T. Thomas II. *OSPF network design solutions*. Cisco Systems, Indianapolis, IN, 1998
184. A. Urpi, M. A. Bonuccelli, and S. Giordano. Modelling cooperation in mobile ad hoc networks: a formal description of selfishness. In *WiOpt 2003*, Sophia-Antipolis, France, 2003. ACM
185. S. Voss. Steiner tree problems in telecommunications. In *Handbook of Optimization in Telecommunications*, pages 459–492. Kluwer, Dordrecht, 2006
186. D. Waitzman, C. Partridge, and S. Deering. Distance vector multicast routing protocol, RFC 1075 – IETF network working group. On-line document: <http://www.ietf.org/>, 1988
187. D. Wall. *Mechanisms for Broadcast and selective broadcast*. PhD thesis, Stanford University, 1980
188. D. Wall. *Mechanisms for broadcast and selective broadcast*. PhD thesis, Computer Science Department, Stanford University, 1982
189. P.-J. Wan, D.-Z. Du, and P. M. Pardalos, editors. *Multichannel Optical Networks: Theory and Practice*, volume 46 of *DIMACS Series*. American Mathematical Society, Providence, RI, 1998
190. C.-F. Wang, C.-T. Liang, and R.-H. Jan. Heuristic algorithms for packing of multiple-group multicasting. *Computers & Operations Research*, 29(7):905–924, 2002
191. W. Wang, X.-Y. Li, and Z. Sun. Share the multicast payment fairly. In L. Wang, editor, *Computing and Combinatorics. 11th Annual International Conference, COCOON 2005*, volume 3595 of *Lecture Notes in Computer Science*, pages 210–19, 2005
192. Z. Wang, S. Zheng, Y. Ye, and S. Boyd. Further relaxations of the semidefinite programming approach to sensor network localization. *SIAM J. on Optimization*, 19:655–673, July 2008
193. B. Waxman. Routing of multipoint connections. *IEEE Journal on Selected Areas in Communications*, 6(9):1617–1622, 1988
194. J. Westbrook and D. Yan. Greedy algorithms for the on-line Steiner tree and generalized Steiner problems. In F. K. H. A. Dehne, J.-R. Sack, N. Santoro, and S. Whitesides, editors, *Algorithms and Data Structures, Third Workshop, WADS '93, Montréal, Canada, August 11–13, 1993*, volume 709 of *Lecture Notes in Computer Science*, pages 621–633, Berlin, 1993. Springer
195. S. Wi and Y. Choi. A delay-constrained distributed multicast routing algorithm. In *Proceeding of the twelfth International Conference on Computer Communication (ICCC'95)*, pages 883–838, San Francisco, CA, 1995. IEEE
196. P. Winter. Steiner problem in networks: a survey. *Networks*, 17:129–167, 1987
197. P. Winter and J. M. Smith. Path-distance heuristics for the Steiner problem in undirected networks. *Algorithmica*, pages 309–327, 1992
198. R. Wittmann and M. Zitterbart. *Multicast Communication, protocols and applications*. Morgan Kaufmann, San Francisco, 1999
199. J. Xue. Graph coloring. In C. Floudas and P. Pardalos, editors, *Encyclopedia of Optimization*, pages 254–254. Springer, 2nd edition, 2009
200. C.-B. Yang and U.-P. Wen. Applying tabu search to backup path planning for multicast networks. *Computers & Operations Research*, 32(11):2875–89, 2005
201. M. Yang, J. Wang, X. Qi, and Y. Jiang. On finding the best partial multicast protection tree under dual-homing architecture. In *Workshop on High Performance Switching and Routing*, pages 128–32, 2005
202. T. S. Yum, M. S. Chen, and Y. W. Leung. Video bandwidth allocation for multimedia teleconferences. *IEEE Trans. on Commun.*, pages 457–465, 1995
203. W. D. Zhong, Y. Onozato, and J. Kaniyil. A copy network with shared buffers for large-scale multicast ATM switching. *IEEE/ACM Transactions on Networking*, 1(2):157–165, 1993
204. Q. Zhu, M. Parsa, and J. J. Garcia-Luna-Aceves. A source-based algorithm for delay-constrained minimum-cost multicasting. In *Proc. IEEE INFOCOM95*, pages 377–385, San Francisco, CA, 1995. IEEE
205. W. T. Zhu. Optimizing the tree structure in secure multicast key management. *IEEE Communications Letters*, 9(5):477–9, 2005



# Index

## Symbols

3SAT, 128

## A

acceptable solution, 101  
acknowledge message, 110  
ad hoc networks, xix, 163  
algorithmic approach, 164  
algorithms, xviii  
approximation algorithm, 98, 137  
approximation algorithms, xviii  
asymmetric links, 165  
asynchronous communications, 101  
asynchronous team, 99  
asynchronous team example, 100  
autonomous heuristics, 101

## B

battery power, 163, 164  
Bellman–Ford algorithm, 59  
broadcasting, 17

## C

cache nodes, 118  
cache placement problem, 120  
cached trees, 49  
capacity, 20  
CBT, 18, 22, 70  
cell phones, 164  
center-based tree, 65  
centroid, 68  
child process, 113  
chromosome, 78  
coloring, 170

combinatorial problems, xviii  
congestion, 72  
connected dominating set, 149  
connected network, 57  
consensus-based heuristics, 102  
construction heuristics, 99, 102  
control messages, 108  
crossover, 79

## D

data items, 168  
delay, 20  
delay constrained Steiner problem, 60  
delay variation, 26  
demand nodes, 47  
destination-driven multicast, 60  
destruction heuristics, 99  
Dijkstra, xviii  
directed graph, 97  
distributed algorithm, 172  
distributed algorithms, 57, 156  
distributed computing, xviii  
distributed messages, 58  
distributed protocol, 59  
dominating set, 149  
driver process, 107  
dummy source, 97  
DVMRP, 22  
dynamic changes, 49  
dynamic groups, 15  
dynamic programming, 98

## E

efficient asynchronous team, 101  
efficient execution, 101

energy expenditure, 163  
Euclidean distance, 149

## F

feasible solution, 96  
flow streaming cache placement, 120  
forest, 96  
fragment, 58

## G

genetic algorithm, 78  
geographical topology, 165  
global connectivity, 59  
global variables, 101, 102  
graph, 20  
GRASP, 81

## H

heuristic, xvii

## I

improvement heuristics, 99, 102  
independent set, 153  
integer optimization, xvii  
integer programming model, 168  
interference, 165  
Internet, xviii  
intra-nets, xviii  
IP formulation, 163

## J

job shop scheduling, 99

## K

KMB heuristic, 58

## L

linear optimization, xvii  
link nodes, 96  
link state advertisement, 10  
load balancing, 122  
local clock, 105  
local improvements, 49  
local optimum, 170  
local perturbation, 170

local search, 59, 170  
local variables, 108  
lower bound, 174

## M

MANET, 151, 164  
MANET topology, 165  
mathematical programming, xviii  
median node, 68  
medical applications, 164  
message exchange, 107  
metaheuristic, 77  
military battlefield operations, 149  
military operations, 164  
minimum connected dominating set, 149  
minimum spanning tree, 59  
mobile clients, 164  
mobile commerce, 149  
MOSPF, 18  
multi-objective optimization, xviii  
multicast dimensioning, 73  
multicast group, xviii, 57, 95  
multicast networks, xviii  
multicast packing, 72  
multicast session, 47  
multicast users, 95  
mutation, 79

## N

neighbors of a node, 20  
network flows, 137  
network hop, 58  
network links, 20  
network traffic, 165

## O

off-line problems, 47  
offspring, 78  
online model, 47

## P

packet routing, xvii  
parallel process, 113  
parallel virtual machine (PVM), 108  
partially synchronous model, 105  
PCADHOC, 164, 168  
peer to peer systems, 164  
PIM, 22, 118  
planar graph, 149  
point-multipoint routing, 95

- point-to-point connection, 95, 97
- polynomial time reduction, 130
- power consumption, 163, 165
- power control, 164
- power control problem, 163
- power level, 167
- power levels, 165
- processing units, 57
- protection tree, 27
- PTAS, 129

## R

- random graph, 50
- remote conferencing, xviii
- rescue groups, 164
- resource allocation, 99
- roulette wheel method, 106
- routing tree cache, 49

## S

- safe message, 110
- satellite connections, 96
- satisfiability, 123
- selection, 79
- set covering, 99
- shared memory, 99–101
- shared multicast tree, 95
- shortest best path tree, 43
- simulated annealing, 81
- software distribution, xviii
- solution population, 99
- sound delivery, 59
- source based tree, 49
- source–destination coupling, 97
- spanning tree, 149
- sparse groups, 15
- speedup, 113
- startup message, 110
- static groups, 15
- Steiner tree, 15, 95
- Steiner trees, xviii
- stream splitters, 120

- streaming caches, 118
- synchronizer module, 109
- synchronous memory, 99

## T

- tabu search, 80
- terminate message, 108
- topological center, 67
- transmission range, 149
- transmission reach, 165, 167
- transmit operation, 96
- transshipment nodes, 96
- traveling salesman, 99
- tree cache placement, 120
- TV system, 47

## U

- undirected graph, 165
- unicast routing, 36
- unit disk graph, 153
- unit-disk graph, 149
- unit-disk graphs, 149
- unity capacity, 125

## V

- variable neighborhood search, 164, 167, 170
- vehicle routing, 96
- video conferencing, 59, 95
- video delivery, 59
- video-on-demand, 21
- virtual backbone, 153
- virtual private network, 119
- virtual-reality, xviii
- VLSI, 96

## W

- web caching, xviii
- wireless networks, 149
- wireless units, 165



## About the Authors

**Carlos Oliveira** obtained a PhD in Operations Research from the University of Florida, a Masters in Computer Science from Universidade Federal do Ceará, Brazil, and a B.Sc. in Computer Science from Universidade Estadual do Ceará, Brazil.

Carlos has spent more than 10 years working on combinatorial optimization problems in several areas, including telecommunications, computational biology, and logistics. He has written more than 20 papers on optimization aspects of these areas. He is an associate editor for *Journal of Global Optimization* and *Optimization Letters*.

Carlos was an assistant professor at Oklahoma State University from 2004 to 2006. Since then he has worked as a consultant in the areas of optimization and software engineering. He works in New York City and lives in New Jersey with his wife and son. Carlos Oliveira can be contacted at his web site <http://coliveira.net>.

**Panos Pardalos** is Distinguished Professor of Industrial and Systems Engineering at the University of Florida. He is also affiliated faculty member of the Computer Science Department, the Hellenic Studies Center, and the Biomedical Engineering Program. He is also the director of the Center for Applied Optimization.

Dr. Pardalos obtained a PhD degree from the University of Minnesota in Computer and Information Sciences. He has held visiting appointments at Princeton University, DIMACS Center, Institute of Mathematics and Applications, FIELDS Institute, AT&T Labs Research, Trier University, Linköping Institute of Technology, and Universities in Greece.

Dr. Pardalos received the degrees of Honorary Doctor from Lobachevski University (Russia) and the V.M. Glushkov Institute of Cybernetics (Ukraine), he is a fellow of AAAS, a fellow of INFORMS, and in 2001 he was awarded the Greek National Award and Gold Medal for Operations Research.

Dr. Pardalos is a world leading expert in global and combinatorial optimization. He is the editor-in-chief of the *Journal of Global Optimization*, *Optimization Letters*, and *Computational Management Science*. In addition, he is the managing editor of several book series, and a member of the editorial board of several international journals. He is the author of eight books and the editor of several

books. He has written numerous articles and developed several well-known software packages. His research is supported by National Science Foundation and other government organizations. His recent research interests include network design problems, optimization in telecommunications, e-commerce, data mining, biomedical applications, and massive computing.